UNIVERSIDAD AUTÓNOMA DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

TRABAJO FIN DE MÁSTER

Sistema Embebido en un Chip para la monitorización remota de temperaturas

Eloy Soto Solana

Septiembre 2013

Sistema Embebido en un Chip para la monitorización remota de temperaturas

AUTOR: Eloy Soto Solana

DIRECTOR: Eduardo Iván Boemo Scalvinoni

Máster en Ingeniería Informática y de Telecomunicación

Escuela Politécnica Superior

Universidad Autónoma de Madrid

Septiembre 2013

Resumen

En el presente trabajo de Fin de Máster se ha analizado, diseñado y desarrollado un prototipo de Sistema Embebido en un Chip para la monitorización remota de temperaturas empleándose sistemas reconfigurables de bajo costo (FPGAs). En concreto, se ha empleado una placa de evaluación "*Spartan-3AN Starter Kit*" del fabricante Xilinx para el desarrollo del citado prototipo.

Sobre el sistema reconfigurable de bajo costo se ha implementado un Microprocesador Soft-Core Microblaze de Xilinx. Los elementos a los que tiene acceso este procesador son una memoria RAM, un interfaz de red Ethernet y un bus de comunicaciones I2C. En este último, se tienen conectados una serie de sensores de temperatura con los que se comunicará nuestro prototipo con el fin de obtener información de los mismos. Estos sensores pertenecen a la familia LM83 del fabricante Texas Instruments, los cuales permiten medir una temperatura local (ofrecida por el propio sensor), y tres temperaturas remotas si se le conectan para ello a las entradas del circuito integrado unos diodos (en este caso, transistores 2N3904).

Una vez obtenida la información sobre las temperaturas, debemos ofrecérsela al usuario para que pueda visualizarla. Para ello, se ha compilado el Kernel de Linux en su versión 3.6 para la arquitectura Microblaze. Con ello, conseguimos tener acceso a una pila de comunicaciones TCP/IP, por lo que se ha diseñado e implementado una aplicación Web que permita visualizar la información recopilada sobre las temperaturas remotas. Esta visualización se lleva a cabo de forma sencilla e intuitiva, pudiendo además de ver el último dato muestreado desde los sensores, un histórico con las últimas temperaturas leídas para monitorizar las variaciones que se producen. Emplear este tipo de aproximación en forma de desarrollo Web nos ha permitido independizarnos del entorno desde el que el usuario desea monitorizar la información, ya que el único requisito será que cuente con un navegador que tenga soporte para el estándar HTML5.

Por otro lado, el prototipo se ha desarrollado de una forma modular, lo que no impide adaptarlo a otros entornos. Con esta premisa, se ha conseguido tener un diseño modular y fácilmente escalable para ser integrado en distintos entornos de operación.

***[Palabras clave]**: System-On-Chip, Microblaze, Linux, I2C, FPGA, Spartan-3*

Abstract

In this final master thesis a System-On-Chip prototype for remote temperature monitoring has been analyzed, designed and developed using low cost reconfigurable logic systems (FPGAs). A "*Spartan-3AN starter kit*" evaluation platform from Xilinx has been chosen for presented prototype implementation.

Over the low cost reconfigurable system a Microblaze Soft-Core microprocessor has been implemented. This processor has access to a RAM memory, an Ethernet network interface and an I2C communications bus. Temperature sensors have been connected to this last element so that our prototype could read temperature data from them. Specifically, LM83 sensors of Texas Instruments manufacturer have been chosen. This integrated circuits offers a local temperature measured from the own chip, and up to three remote temperatures if we connect diodes to its input pins (in this case, we have used 2N3904 transistors for this purpose).

Once remote temperatures are obtained from these sensors, we have to show this data to the user. For this, a GNU/Linux kernel (3.6 version) has been compiled for Microblaze architecture. Running such operative system offers us TCP/IP connectivity, so a Web application has been designed and developed in order to show obtained data to final user. This visualization is carried out in a simple and intuitive way, displaying either last obtained temperature and historical of each connected sensor so we can visualize produced variations over time. Using a Web based implementation allow us to be independent in which final system is chosen by the user to visualize data, being the only requisite that a Web navigator with HTML5 support is used.

On the other hand, presented prototype has been developed in a modular way, being easily adaptable to other environments. With this premise, we have obtained a modular design easily scalable for being integrated in different environments of operation.

***[Keywords]**: System-On-Chip, Microblaze, Linux, I2C, FPGA, Spartan-3*

Agradecimientos

Cada vez que termino una etapa como estudiante y tengo que escribir agradecimientos, en primer lugar y sin lugar a dudas van a mis padres. Gracias por vuestros continuos esfuerzos. Con este van tres, y los tres os los debo a vosotros.

A mi hermana, como no, por estar siempre ahí echando una mano y animándome.

A mis amigos por la comprensión, los ánimos, la preocupación y el apoyo. Por las risas y los buenos momentos, y por la paciencia. Sin vosotros hubiese sido infinitamente más duro.

A mis compañeros del máster. Gracias por hacer de esta etapa algo agradable para recordar. En especial a Pablo, por tantos buenos momentos, por la ayuda y la preocupación. Muchas gracias ratón. Gracias a Rodolfo-Francisco [RF] por todas las risas que nos hizo pasar.

También quiero darle las gracias a Eduardo, como no, por su guía a lo largo de estos dos años. Por estar siempre dispuesto a echar una mano, y por contar conmigo.

Y creo que es hora de agradecérmelo a mí mismo también, que lo mío me ha costado.

Índice de contenidos

Índice de ilustraciones

Índice de tablas

1 Introducción

En este Proyecto de Fin de Máster se describe el empleo de lógica reconfigurable de bajo costo para materializar un sistema para la monitorización remota de temperaturas. Una de las premisas del presente desarrollo es que el prototipo obtenido sea adaptable a distintos entornos con un mínimo de modificaciones en el mismo.

El marco tecnológico será la familia Spartan-3 de Xilinx [DR1], en concreto la placa de evaluación "*Spartan-3AN Starter Kit*" [DR2]. Adicionalmente, se analizará el estado del arte de algunas investigaciones previas que emplean Linux sobre FPGAs, como por ejemplo [DR3] [DR4] y [DR5].

Para desarrollar este prototipo, se ha implementado un sistema que se adecua, a grandes rasgos, al siguiente esquema:

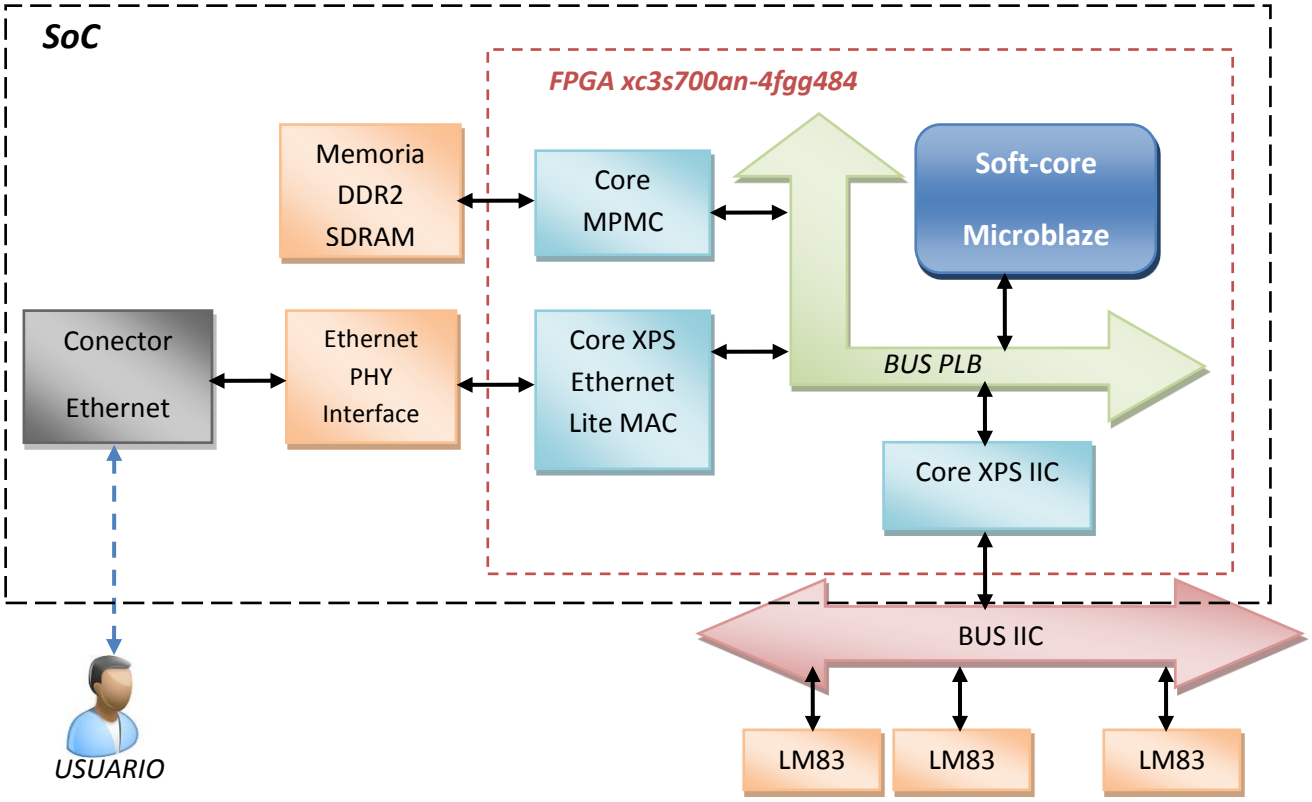


Ilustración 1: Esquema general del Sistema Embebido desarrollado

En el entorno reconfigurable se ha sintetizado un procesador soft-core Microblaze de Xilinx. Además, se empleará la lógica disponible para implementar varios periféricos, entre los que podemos destacar un controlador de memoria multipuerto (MPMC), un periférico para obtener el nivel MAC de la arquitectura de comunicaciones TCP/IP (Ethernet Lite MAC) y un controlador para poder acceder a un bus de comunicaciones I2C (XPS IIC).

Sobre el procesador Microblaze de Xilinx se ha desplegado un sistema operativo de la familia GNU/Linux. Esta capa nos permitirá tener todas las facilidades que ofrece un sistema operativo, incluyendo conectividad TCP/IP. Para esta parte del proyecto se analizarán una serie de documentos, partiendo de la documentación oficial de Xilinx [DR6] sobre Linux en sus procesadores tanto soft-core como hard-core. Se emplearán también una serie de tutoriales y presentaciones oficiales [DR7] [DR8] [DR9] para ampliar la información relativa a ejecutar Linux sobre Microblaze. También será necesario analizar las características del procesador [DR10] y las opciones que tenemos disponibles para el desarrollo de Software sobre el mismo tanto sin el sistema Linux [DR11] como con Linux corriendo por encima [DR12].

Por otra parte, nuestro sistema tendrá acceso al bus de comunicaciones I2C [DR13]. Para tener acceso al mismo desde el sistema operativo se debe comprender cómo se implementas los drivers en los sistemas operativos GNU/Linux [DR14] ya que, además de tener el periférico hardware, este debe de ir acompañado de un driver que nos permita comunicarlos con él desde la aplicación que se desarrolle, y deberemos analizar también la forma en que se implementan los sistemas Linux embebidos [DR15].

La información recopilada se mostrará al usuario mediante una conexión Ethernet, mediante una aplicación Web, de tal forma que un usuario cualquiera pueda acceder a esta información empleando un navegador Web, lo que amplía el margen de sistemas operativos que pueden utilizar los clientes, y ofrece una gran facilidad de acceso a la información empleando únicamente un navegador.

Generalizando, se desea obtener un prototipo que sea adaptable a varios entornos, centrándose en su implementación sobre sistemas reconfigurables de bajo coste, y que permita acceder a la monitorización remota de temperaturas mediante un bus de comunicaciones I2C. Además, se desea que la facilidad con la que los usuarios pueden acceder y visualizar estos datos presente un alto grado, y que esta monitorización pueda llevarse a cabo de forma remota.

Como podemos ver, este proyecto encaja perfectamente en la especialidad de "Sistemas Embebidos y Reconfigurables" de los estudios de Máster. Por un lado, se profundiza en las temáticas de Sistemas Embebidos en un Chip y en el Codiseño Hardware/Software y, por otra parte, tenemos también relacionada la parte de los estudios que cubre el apartado de Sistemas Operativos en entornos Embebidos.

1.1 Motivación

Los Sistemas Embebidos en un Chip (SoC, System-on-Chip) resultan elementos que otorgan una alta capacidad de parametrización. Al implementarse sobre sistemas de lógica reconfigurable (FPGAs), el ámbito de entornos y situaciones a los que pueden adaptarse crece exponencialmente. Tener un procesador como elemento central en un diseño y, a su alrededor, lógica reconfigurable nos da la opción de poder diseñar un sistema embebido acorde a las necesidades concretas que se nos planteen, adaptándolo a la perfección al

problema a resolver. Por otro lado, contar con un entorno en el que el hardware que rodea a nuestro microprocesador puede ser modificado sin necesidad de fabricar elementos a medida (ASICs) redunda en un gran ahorro en costes de prototipado y producción. También esta premisa nos ofrece le opción de poder modificar nuestro diseño para funcionar con un mismo elemento central (procesador) sobre distintos tipos de periféricos conectados al mismo.

Precisamente estas premisas son las que motivaron la presente Tesis de Fin de Máster. Esta gran capacidad de adaptación y modularización de grano fino hizo que surgiese la idea de emplearla para llevar a cabo un prototipo viable que se adaptase a un problema planteado.

Por otro lado, las capacidades que han ido adquiriendo las FPGAs de bajo coste, tal y como podemos ver en la siguiente imagen [16] hacen que resulten cada vez más propicias para llevar a cabo desarrollos de esta índole:

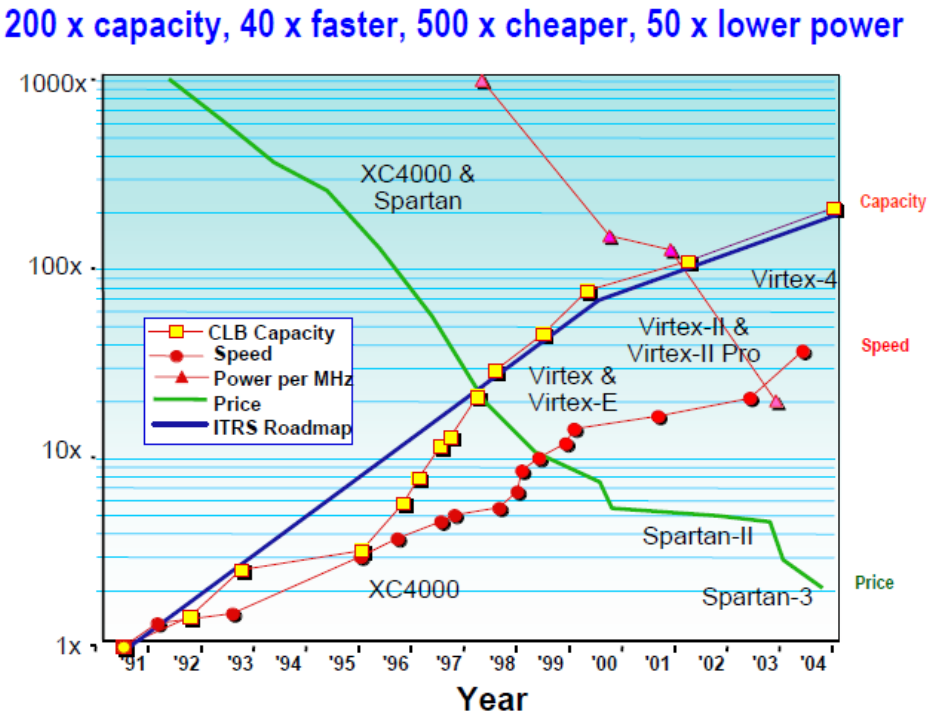


Ilustración 2: Evolución de las familias de FPGAs de bajo coste de Xilinx desde 1990 a 2005

De la gráfica anterior podemos deducir que los sistemas más asequibles del fabricante Xilinx han reducido su coste en prácticamente un 1000%, aumentando si capacidad en más de un 100% y su velocidad en prácticamente un 50%. Además, el consumo por MHz que se produce ha descendido drásticamente. Estas situaciones las hacen cada vez más idóneas para integrarse en entornos embebidos, donde el consumo, el rendimiento y el precio resultan elementos fundamentales.

Otra argumentación del peso que están cogiendo este tipo de componentes en los sistemas embebidos la tenemos en la aparición del soporte oficial para la arquitectura del Soft-Core Microblaze de Xilinx a partir de la versión 2.6.31 del Kernel de GNU/Linux [17] [18].

Así pues, con todo lo expuesto, el presente Trabajo de Fin de Máster pretende ahondar en este mundo que se encuentra en creciente expansión. Se consolidarán y ampliarán conocimientos relacionados con la especialidad de Sistemas Embebidos y Reconfigurables de los estudios de máster realizados, en concreto profundizando en materias como los Sistemas Embebidos en un Chip, los Sistemas Operativos Embebidos (concretamente en entornos reconfigurables), y el Codiseño Hardware/Software.

1.2 Objetivos

Los objetivos más concretos que se persiguen con el siguiente Trabajo de Fin de Máster de cara a conseguir el objetivo general del prototipo propuesto son:

1. **Investigar el despliegue del Sistema Operativo GNU/Linux sobre entornos reconfigurables de bajo coste**: Para ello, se ha seleccionado una placa de evaluación *Spartan-3AN Starter Kit*. En este aspecto, se analizará la forma en que podemos ejecutar el Sistema Operativo sobre Microblaze en esta FPGA. No se ha encontrado ninguna referencia que explique cómo desplegar el sistema en este modelo en concreto, por lo que se verá la viabilidad y necesidades que se presentan para poder conseguirlo. Así mismo, se profundizará en cómo debe configurarse y construirse el Kernel de Linux para poder ejecutarse sobre este soft-core, pues existen soluciones comerciales para el mismo (como *PetaLinux* [DR19]) que no se van a usar en este proyecto, si no que se obtendrá el código fuente del Kernel, se configurará y se compilará para su correcto funcionamiento. Por otro lado, este punto aporta un valor añadido al demostrar que emplear esta placa para desplegar Linux es viable, pues Xilinx sólo emplea los kits de evaluación *SP605*, *ML507* y *Zynq-7000* como entornos de pruebas no asegurando el correcto funcionamiento en otros distintos. Además de lo anterior, se podrá contribuir al estado del arte en este aspecto, pues la gran mayoría de referencias que se han encontrado en este apartado son relativamente antiguas, detallando el uso de las versiones del Kernel 2.6 y cercanas, mientras que en este Trabajo se empleará la versión 3.6, aportando pues una actualización en este aspecto a la información existente.

2. **Acceso al bus de comunicaciones I2C desde una aplicación Software corriendo sobre el Sistema Operativo GNU/Linux**: De nuevo, este apartado proporciona una actualización del estado del arte. En la documentación oficial de Xilinx se detalla cómo acceder a una memoria EEPROM conectada a un bus I2C [DR20] en la placa de evaluación *ML507*, no detallando ningún otro dispositivo. Por lo tanto, en este TFdM se accederá a sensores de temperatura LM83 de Texas Instruments, con lo que se demostrará la viabilidad de obtener información de otros dispositivos conectados a este bus, generalizando el método de acceso al mismo. Tampoco se han encontrado referencias sobre acceso a estos elementos desde una FPGA de Xilinx corriendo Linux sobre el soft-core Microblaze, por lo que de nuevo se aporta información en este ámbito.

3. **Configuración de un servidor Web embebido**: Una vez tengamos acceso a la información sobre las temperaturas, este debe de ser mostrada al usuario de forma

remota. Para ello se empleará una aplicación Web, por lo que debe configurarse un servidor Web embebido para poder conseguir este objetivo.

4. **Definición de las interfaces entre la aplicación que accede al bus I2C y la que mostrará esa información al usuario**: Como ya hemos dicho, los datos de temperaturas se mostrarán a través de una aplicación Web. Otro de los objetivos es ver las distintas posibilidades que tenemos para comunicar la aplicación que acceda a los sensores y la que muestre esos datos al usuario, seleccionando la mejor opción y definiendo el interfaz definitivo para la comunicación entre ambas.

5. **Desarrollo de la aplicación Web que muestre la información al usuario remoto**: Otro de los objetivos es desarrollar y probar la aplicación Web que muestre la información sobre temperaturas al usuario final.

Como vemos, todos los objetivos anteriores conducen al desarrollo final de un prototipo de Sistema Embebido en un Chip que permita llevar a cabo la monitorización remota de temperaturas, que es el hito general del presente TFdM.

1.3 Estructura del documento

La presente memoria de TFdM consta de 7 capítulos estructurados de la siguiente forma:

1. En el primer apartado se lleva a cabo una introducción al proyecto realizado, la motivación que dio lugar al mismo, los objetivos que se persiguen y se presenta de forma global cómo está estructurado este documento.

2. El segundo apartado expone el estado del arte en cuanto a los temas que se encuentran relacionados con este proyecto, exponiendo investigaciones llevadas a cabo que emplean el despliegue de sistemas operativos GNU/Linux sobre Sistemas Embebidos en un Chip empleando para ello FPGAs. También, se dará un pequeño repaso a la evolución del bus de comunicaciones I2C.

3. El tercer apartado expone las consideraciones a tener en cuenta a la hora de llevar a cabo el presente proyecto. Para ello, se comenta el entorno de desarrollo necesario, las herramientas y su configuración, etc. También se expondrán puntos clave y necesidades que no deben ser obviadas, por ejemplo a la hora de compilar el Kernel de GNU/Linux para una arquitectura como la que se ha empleado. También se expondrá el desarrollo de una pequeña aplicación de ejemplo que sirva como base para la generación de Software de forma cruzada.

4. Una vez que se han visto las necesidades en cuanto al desarrollo se refiere, en el cuarto apartado se expone la arquitectura Software que se ha diseñado e implementado para conseguir los objetivos finales perseguidos con este trabajo. Se

analizan los requisitos de alto nivel que debe de cumplir la aplicación, se da una visión sobre el entorno tecnológico en el que se desplegará, se describe el funcionamiento de la aplicación y se muestran los modelos de diseño de la misma, como son el diseño de datos, los diagramas de clases, etc.

5. En el quinto apartado se presentan las conclusiones obtenidas tras la realización del presente proyecto.

6. El sexto apartado contiene posibles vías de trabajo futuro en las que se podría seguir desarrollando y ampliando el prototipo expuesto.

7. En el séptimo epígrafe se expondrá la planificación del proyecto, así como la dedicación que se ha llevado a cabo.

8. El octavo apartado expondrá las referencias bibliográficas empleadas durante la realización de este proyecto.

9. Por último, se ha incluido un anexo que no iba a ser parte del presente proyecto, pero que en la última semana antes de finalizar la redacción de la presente memoria se ha llevado a cabo. En él, se expone como agregar un reloj de tiempo real (RTC) al bus de comunicaciones I2C, y cómo se debe configurar el Kernel del sistema operativo para que reconozca este elemento durante su arranque y sea así capaz de obtener la fecha y hora desde el mismo para configurar el sistema.

2 Estado del arte

El presente apartado pretende servir de antecedente respecto al estado del arte de los aspectos más importantes involucrados en el presente Proyecto de Fin de Máster.

2.1 Linux embebido en sistemas reconfigurables

La gran proliferación del sistema operativo GNU/Linux radica en su naturaleza *open source.* El poder acceder al código de este sistema operativo libremente para adaptarlo, modificarlo y ajustarlo a nuestras necesidades hace que sea la opción por excelencia empleada en los desarrollos de sistemas embebidos. En nuestro caso concreto, tenemos un entorno de lógica reconfigurable sobre el que se implementará un Soft-Core Microblaze. Este procesador presenta una arquitectura que es lo esencial para hacer correr un sistema operativo sobre él. Tener soporte para este procesador en el Kernel de Linux hace que se puedan abordar una gran cantidad de tareas de una forma mucho más sencilla y asequible que si nos encontrásemos ante un enfoque puramente hardware.

Esta situación ha propiciado que los entornos reconfigurables se apliquen a una creciente cantidad de áreas en las que antes resultaba muy difícil encajarlos. Un campo en el que se encuentran altamente extendidos es en los entornos de red. Por ejemplo, emplear sistemas basados en FPGAs corriendo por encima un sistema operativo de la familia GNU/Linux permite tener una serie de servicios Software como un sistema gestor de bases de datos, un servidor Web, y servicios como SNMP para llevar a cabo un proceso de monitorización de red [DR21], recolectando estadísticas sobre el consumo de ancho de banda o monitorizando actividades sospechosas en la misma con el fin de detectar posibles incursiones no autorizadas.

También en este ámbito encontramos investigaciones que emplean las bondades de tener hardware reconfigurable y posibilidades de ejecutar un sistema operativo con los consecuentes servicios que ofrece. En [DR22], por ejemplo, se expone una aproximación en la que se desacoplan las funciones típicas del reenvío de paquetes que se producen en el nivel 2 de la arquitectura OSI, al control de flujo y temas más computacionales en cuanto a nivel de protocolos que se debe de realizar. Para el reenvío, se emplea una implementación en Hardware, de forma que se consiga un mayor ancho de banda y velocidad de proceso, mientras que a la capa Software que corre sobre Linux se despliega del plano del control de tráfico. Siguiendo este enfoque, soluciones como las expuestas en [DR23] emplean las capacidades Hardware de una FPGA para llevar a cabo tareas de rutado y reenvío de paquetes, mientras que en el plano del Software delegan tareas de filtrado de información empleando IPTables sobre Linux, consiguiendo así un control de grano fino sobre la información que circula por la LAN y aplicando medidas preventivas para minimizar ataques sufridos desde una WAN.

Otros ámbitos en los que se da el uso de SoC que corren sistemas Software en campos de la robótica. Por ejemplo, en [DR24] se expone un sistema para control de robots industriales. Este sistema aventaja a los clásicos PCs de control industrial en coste, espacio y consumo. Mediante el plano Software, se ofrece al usuario la posibilidad de controlar los elementos finales (como por ejemplo, brazos robóticos), mientras que en el plano Hardware se han implementado los filtros necesarios, algoritmos de visión, controladores de motor, etc. En cuanto al campo de robots autónomos, en [DR25] se exponen los resultados de emplear la unión del Hardware y el Software para llevar a cabo procesamiento de imagen en tiempo real. Conectando un sensor CMOS a este sistema, se han implementado algoritmos de procesamiento de imagen directamente en el Hardware para llevar a cabo esta tarea de forma más rápida, y se ha delegado en la capa Software la implementación de algoritmos de seguimiento de objetos en movimiento y de reconocimiento espacial.

Más aplicado al propio campo de los sistemas reconfigurables tenemos el área de la reconfiguración parcial dinámica de dispositivos lógicos reprogramables (FPGAs). Esta capacidad permite cambiar partes del diseño Hardware al vuelo, es decir, reprogramar partes del dispositivo descargando sobre ellas otro bitstream, de forma que implementen un nuevo periférico/funcionalidad completamente distinta, y sin afectar al funcionamiento del resto de elementos que operan en la FPGA. En [DR26] se expone un sistema de reconfiguración parcial dinámica que permite llevar a cabo esta tarea a través de Internet, de forma remota. Para ello, se usan los elementos Hardware que ofrecen las FPGAs de Xilinx y que permiten realizar esta configuración parcial dinámica (puertos ICAP), y se despliega un sistema Linux sobre un procesador PowerPC que ofrecerá conectividad a la red. Así, los distintos bitstream que se cargarán en las áreas de la FPGA pueden ser descargados a la placa de forma remota, y el Software desarrollado se encarga de proporcionar ese bitstream al elemento Hardware que reprogramará parcialmente el dispositivo. Este enfoque ofrece una gran ventaja, y es el acceso remoto a elementos a los que, una vez integrados en su entorno final, resulta muy difícil acceder de forma física.

Otras aplicaciones de este tipo de sistemas sugieren desarrollos como por ejemplo el presentado en [DR27], en el que se ha desarrollado un core SATA2 de naturaleza *open source*, y un driver igualmente libre que puede ser agregado a un Kernel de Linux para ofrecer acceso a estos discos desde el sistema operativo. Este caso en concreto me parece muy interesante ya que uno de los problemas que se daban en FPGAs y sistemas operativos corriendo sobre ellas es, por un lado la poca capacidad de almacenamiento que se suele tener, y por otro la escasa velocidad que presentan estos medios de almacenamiento. Tener acceso a discos, por ejemplo SSDs, desde el sistema Linux abre un abanico de posibilidades en las que el cuello de botella que eran los HDDs se minimiza considerablemente.

Como vemos, el número de campos en los que emplear Sistemas Embebidos en un Chip que emplean tanto elementos Hardware como Software toma cada día más relevancia. Por ello, este proyecto ha querido centrarse en este tipo de sistemas, diseñando un prototipo que monitoriza temperaturas y permite visualizarlas de forma remota. Con ello se pretende aportar un nuevo campo de aplicación a esta área en constante crecimiento.

2.2 El bus de comunicaciones I2C

Inicialmente, el BUS I2C fue desarrollado por Philips Semiconductors en 1982 para interconectar una CPU y los periféricos que la rodeaban en un sistema de televisión [DR28]. Sin embargo, este bus fue creciendo en popularidad debido a su sencillez ya que emplea un protocolo de comunicaciones serie muy simple haciendo uso únicamente de dos líneas de comunicación: SDA (Serial Data) para transmitir la información, y SCL (Serial CLock) para manejar una señal de reloj que permita el sincronismo y temporización. Este bus es básicamente una topología de tipo maestro-esclavo, en la que el maestro va haciendo solicitudes de información a los distintos esclavos conectados al bus, y estos responden ante estas solicitudes. El direccionamiento se lleva a cabo mediante palabras de 7 bits, más el octavo que representa un bit de lectura/escritura, aunque existe un modo de direccionamiento extendido que permite emplear 10 bits con el fin de poder tener más periféricos conectados al bus.

Debido sobre todo a esta sencillez, la proliferación que ha sufrido este bus de comunicaciones ha sido muy alta. En 1992 Philips lanzó la especificación de la versión 1.0. En ella se eliminó el modo en que venía funcionando el bus hasta entonces (low-speed), y se añadió el denominado modo fast-speed, que permitía velocidades de transmisión de hasta 400 kbit/s.

En 1998 ve la luz la especificación 2.0 de este protocolo. En ella, las principales novedades introducidas fueron la inclusión de un nuevo modo de alta velocidad (que permite alcanzar hasta 3.4 Mbit/s) y la inclusión de niveles de voltaje en las líneas del bus en vez de valores fijos a las entradas.

En 2007 se publica la versión 3.0 de este protocolo, en el que la principal novedad que presenta es elevar la tasa de transmisión de sistemas que operaban hasta 400 kbit/s hasta 1 Mbit/s.

Finalmente, en Febrero de 2013 se presentó la versión 4.0 de este extendido protocolo de comunicaciones. En ella (última versión existente a fecha de escritura de este documento), se presenta un nuevo modo de funcionamiento con bus unidireccional y velocidades de transmisión que alcanzan hasta 5 Mbit/s.

Como se puede observar, la evolución de este protocolo ha sido bastante notable. Se ha ido mejorando con el fin de obtener mayores tasas de transferencia. El presentar en la última versión un sistema que se basa en comunicación unidireccional nos lleva a pensar en aplicaciones en las que ya no tengamos un maestro y varios esclavos como tal, sino en un nodo central al que los distintos elementos periféricos irán enviando datos. Esto permite que las tasas de transferencia puedan elevarse (de ahí llegar a los 5 Mbit/s) ya que las comunicaciones se simplifican.

Por su sencillez y facilidad de implementación (el bus son únicamente dos líneas a las que hay que conectar unas resistencias de pull-up entre la línea y Vcc), la proliferación de distintos tipos de elementos compatibles con este bus es cada vez mayor. Entre las distintas

aplicaciones que puede tener, se puede acceder a memorias EEPROM, SDRAM, DDR SDRAM, etc. También a memorias no volátiles, ADCs y DACs de baja velocidad, paneles LCD/OLED, relojes en tiempo real, sensores de temperatura, altitud, inclinación, humedad, etc. Esto hace que el número de elementos de los cuales se puede obtener información sea muy grande.

Si nos centramos en los sistemas embebidos, estos buses de comunicación tipo serie resultan ideales. Por un lado, son muy sencillos de manejar, por lo que la carga computacional que conlleva el realizar comunicaciones a través de ellos es muy baja. Por otro lado, se emplean únicamente dos líneas de comunicación, por lo que el número de recursos físicos necesarios (por ejemplo, pines de E/S) para su conexión es mínimo. Con estos pocos recursos empleados, tenemos acceso a una gran cantidad de elementos diferentes que nos ofrecen posibilidades de almacenamiento, monitorización, conversión de datos, etc.

En el caso que nos ocupa, este bus de comunicaciones ha sido empleado para conectar al mismo sensores de temperatura LM83 del fabricante Texas Instruments, de forma que se pueda acceder a la información que proporcionan los mismos.

3 Entorno de trabajo para el desarrollo del Sistema Embebido en un Chip expuesto en el presente proyecto

En este apartado expondremos el entorno de trabajo necesario y su configuración para el desarrollo del presente proyecto. Para el apartado de implementación sobre la FPGA, necesitaremos contar con una herramienta fundamental, el software EDK de Xilinx. Mediante este sistema podremos modelar nuestro System-on-Chip sobre el que más tarde ejecutaremos Linux y el Software desarrollado. Cabe destacar que el sistema operativo sobre el que se suelen llevar a cabo este tipo de desarrollos, típicamente, es alguna distribución GNU/Linux debido a la gran cantidad de herramientas con las que cuenta. Para el desarrollo del presente proyecto, se ha empleado una distribución Ubuntu 10.04 LTS *Lucid Lynx*, por lo que todo el proceso se lleva a cabo sobre este sistema, con las versiones de las herramientas disponibles para el mismo.

3.1 Esquemático del circuito para las pruebas de medición de temperaturas

Nuestro sistema se va a centrar en acceder a un bus de comunicaciones I2C para obtener datos de temperaturas de unos sensores conectados al mismo. Para poder desarrollar el prototipo descrito en este documento, tenemos que contar con un montaje de un circuito que cuente con estos elementos para poder verificar el funcionamiento del sistema en su conjunto. Para ello, en la siguiente imagen podemos ver un esquemático que muestra este montaje:

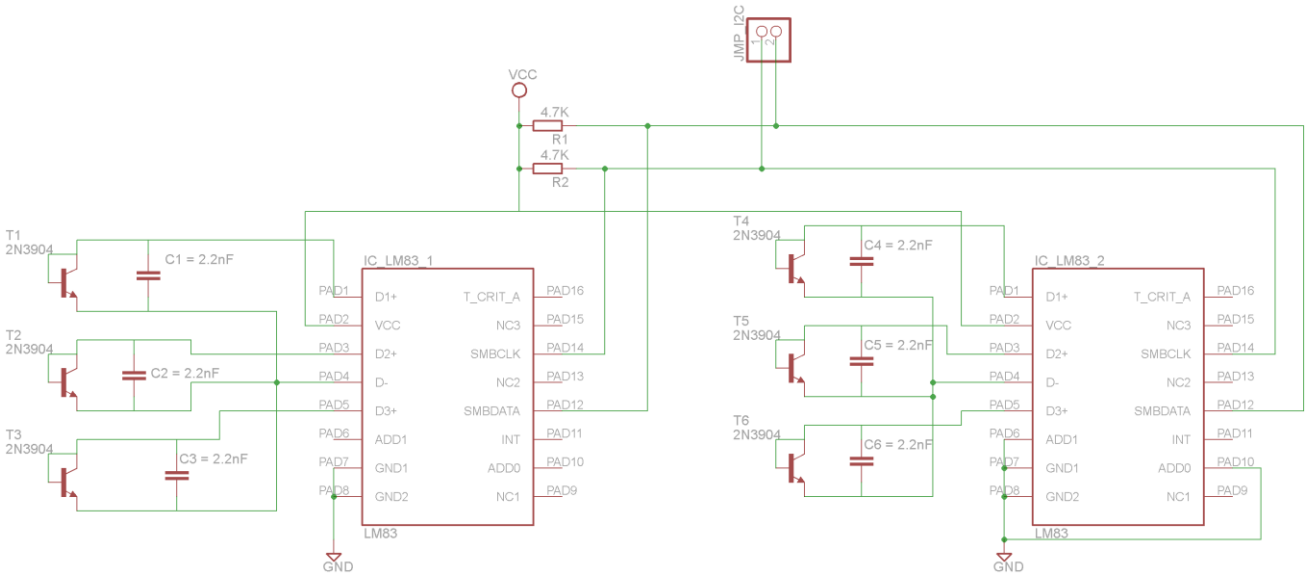


Ilustración 3: Esquemático del circuito para las pruebas de medición de temperaturas

Como podemos observar, en el esquemático se han empleado transistores NPN cortocircuitando base y colector para que se comporte como un diodo de temperatura. Un transistor de este tipo puede ser visto como dos diodos que son controlados por la base. En una configuración típica, entre base y emisor tenemos un diodo polarizado en directa y entre base y colector uno en inversa.

Otro punto a tener en cuenta son los condensadores para los transitorios introducidos entre los dos terminales del diodo. Estos deben encontrarse lo más cerca posible del sensor LM83 [DR29]. También debemos detallar que las direcciones del bus I2C para los integrados LM83 pueden seleccionarse mediante los pines ADD0 y ADD1, que dependiendo del estado de los mismos, cada sensor tomará una dirección distinta. En este caso, el primer LM83 tiene los pines sin conectar, y el otro puenteados a masa. Es importante tener en cuenta que esta dirección se mantendrá siempre fija mientras los sensores se encuentren conectados a una fuente de alimentación aunque se cambie el valor de estos pines [DR29], por lo tanto su dirección no cambiaría hasta que no se apagase su alimentación y se volviese a conectar.

Por último, vemos que las líneas SDA y SCL están conectadas a Vcc mediante unos resistores de pull-up, y de ellas sacamos un conector de dos pines para llevarnos este bus hasta la placa de evaluación mediante unos cables. Otro dato a remarcar en este aspecto es que si alimentamos este circuito de forma independiente, debemos unir su masa junto a alguna de la que nos ofrece la FPGA en el banco de I/O de la misma donde ubiquemos los pines del bus. Se recomienda la alimentación por separado en vez de desde la propia placa de evaluación para evitar que exista una demanda de corriente excesiva y que los reguladores de la placa de evaluación se estropeen por no poder satisfacerla. Es improbable que esto pase, y más teniendo en cuenta que en estas placas suele haber varios reguladores para alimentar distintas zonas de la FPGA, por lo que la corriente que demande el circuito externo añadida a la de los propios elementos de la placa queda repartida por estos distintos reguladores, pero también puede darse el caso de que varios elementos se alimenten desde el mismo regulador (recordemos que en este diseño se emplea conectividad Ethernet, memoria SDRAM DDR2, puerto serie, …), por lo que podría suceder que la demanda fuese demasiado alta. Para comprobar este supuesto, se podría mirar el esquemático de la placa de evaluación y ver cómo se reparten las alimentaciones para deducir si la carga está repartida o centralizada, y tomar la decisión de si alimentar el circuito desde la placa de evaluación o no, pero en el presente prototipo se ha optado directamente por montar un circuito externo de alimentación basado en un regulador de tensión LM317T.

3.2 Descargando los entornos repositorios necesarios para el desarrollo

Ya que una parte de nuestro objetivo es compilar el Kernel de Linux para la arquitectura Microblaze y luego desarrollar aplicaciones que corran sobre él, necesitaremos contar con una serie de elementos para esta tarea:

- **Toolchain**: Se denomina *toolchain* a un conjunto de herramientas que nos permiten el desarrollo desde una arquitectura (Intel i386 en este caso) para obtener programas que serán ejecutados sobre una arquitectura completamente distinta (Microblaze en el presente proyecto). Para ello, necesitamos una serie de programas (como compiladores, ensambladores, linkers, etc.), que generen código ejecutable para la máquina destino, pero que puedan ser lanzadas (las herramientas) en la arquitectura de origen. A este paquete de utilidades, en el mundo del desarrollo de sistemas embebidos, se le denomina *toolchain*.

- **Código fuente del Kernel de Linux**: Por otro lado, necesitaremos el código fuente del Kernel de GNU/Linux. Este código será compilado mediante el *toolchain* empleando uno de sus compiladores cruzados para generar el binario del Kernel que pueda ejecutarse sobre Microblaze.

- **IDE de desarrollo**: Esta herramienta no es imprescindible, pero nos ayuda bastante durante el proceso de desarrollo. Para la realización del presente proyecto se ha empleado el IDE Eclipse Juno para Linux.

3.2.1 Obteniendo el toolchain y el código fuente del Kernel de Linux

Existen varias maneras para obtener el código fuente del Kernel de GNU/Linux y un *toolchain* que nos permita desarrollar para Microblaze. Sin embargo, Xilinx pone a nuestro alcance unos repositorios desde los que podemos descargar ambos elementos.

Estos repositorios se encuentran mantenidos mediante un sistema de control de versiones denominado GIT. No es objeto de este documento explicar ni cómo instalar este entorno ni cómo funciona el mismo, pero básicamente GIT es un sistema de control de versiones distribuido (a diferencia de otros como por ejemplo CVS o Subversion). Es decir, descargaremos una copia del código sobre el que vamos a trabajar (denominada repositorio), y podremos llevar a cabo las modificaciones y acciones que deseemos sin que el origen se vea afectado (nos permite trabajar sin conexión a un servidor central que controle los cambios, de ahí que sea distribuido). Típicamente, una vez que los cambios que hemos realizado sean definitivos, los subiremos al servidor para que estén disponibles para otros participantes en el proyecto.

Una vez instalado un cliente de GIT podemos acceder a los repositorios que Xilinx nos ofrece tal y como se indica en el apartado "*Development environment → Development Host Setup → Microblaze GNU Tools Description*" de la Web [DR6]. Para obtener el *toolchain*, nos clonaremos el repositorio "microblaze-gnu.git". Para acceder a las fuentes del Kernel que nos ofrece Xilinx (es mejor que las oficiales, ya que aplican actualizaciones para sus sistemas antes de que estas se reflejen en el mainline oficial, y mantienen drivers que no podemos encontrar en el Kernel original), debemos de clonarnos el repositorio "linux-xlnx.git".

Con lo comentado anteriormente, ya tenemos lo esencial para compilar el Kernel y nuestras aplicaciones para la arquitectura objeto Microblaze. Sin embargo, también necesitamos otra herramienta que más tarde será integrada con EDK. Es el "Device Tree Generator". Esta herramienta nos permitirá generar el fichero DTS para nuestro sistema embebido.

Un fichero DTS (Device Tree Source) contiene una descripción del Hardware de nuestro sistema embebido. Ya que nos encontramos ante sistemas reconfigurables que pueden implementar una multitud de elementos hardware diferentes, compilar esta información junto al Kernel no resulta útil, pues ante un cambio tendríamos que estar recompilando todo el sistema. Para ello, se usa un mecanismo en el que el fichero DTS que describe nuestro hardware se "compila" en un fichero DTB (Device Tree Blob), y este es leído por el Kernel de Linux al arrancar el sistema. Esto también es necesario debido a que nuestro entorno de desarrollo (FPGAs) no cuentan con la estructura típica de un PC, en el que durante el arranque existe un pequeño código ejecutable (la BIOS, Basic Input/output System) que se encarga de analizar el Hardware que tenemos conectado, lleva a cabo unas comprobaciones rutinarias sobre el mismo, y pasa esa información al Sistema Operativo al inicio de su ejecución. Por este motivo, emplear ficheros DTS en el desarrollo sobre FPGAs resulta idóneo, máxime si tenemos en cuenta que pueden existir periféricos propios y no estándar, que no podrían ser reconocidos directamente por un sistema tipo BIOS.

Para obtener esta herramienta, debemos clonarnos el repositorio "device-tree.git" que nos ofrece Xilinx. Sin embargo, este caso es especial, ya que en nuestra copia local, este debe de ajustarse a una estructura de directorios concreta para poder ser reconocido por EDK. La estructura que contenga el repositorio debe ser "device_tree_repo/bsp/device_tree_v0_00_x". Una vez clonado de esta manera, debemos de asegurarnos de que dentro de la última carpeta, existe otra dentro llamada "data", y dentro de esta los ficheros "device_tree_v2_1_0.tcl" y "device_tree_v2_1_0.mdl", tal y como podemos ver en la siguiente imagen:

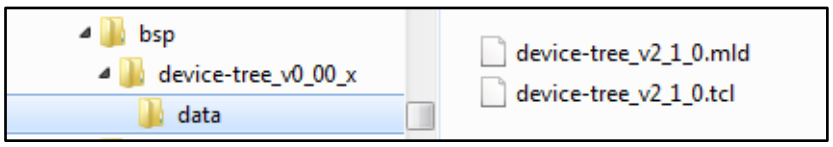


Ilustración 4: Estructura del repositorio local para albergar el Device Tree Generator de Xilinx

3.2.1.1 Variables de entorno para la compilación cruzada

Una vez descargados los repositorios, ya podríamos compilar el Kernel de Linux (configurándolo previamente, lo cual veremos en otro apartado) y nuestras aplicaciones para la arquitectura Microblaze. Sin embargo, debemos de agregar dos variables de entorno para no tener que estar incluyéndolas cada vez que deseemos realizar esta tarea. Lo ideal es agregarlas al fichero '.bashrc' de nuestro directorio de trabajo, de forma que cada vez que

lancemos una Shell 'bash' se carguen automáticamente. Las dos entradas a dar de alta en este fichero serían:

1. export PATH="*RUTA_A_NUESTRO_REPOSITORIO*"/microblaze-gnu/binaries/lin32-microblaze-unknown-linux-gnu_14.3_early/bin/:$PATH
2. export CROSS_COMPILE=microblaze-unknown-linux-gnu-

De esta forma, con la primera entrada agregamos la ruta a los binarios del toolchain y con la segunda, el prefijo que se empleará por el sistema 'make' al llevar a cabo el proceso de compilación del Kernel para Microblaze (hacemos notar que no empleamos la versión 'le' de las herramientas, es decir, hemos seleccionado la implementación 'big-endian' de Microblaze).

3.3 Diseño del sistema embebido

Para el desarrollo del sistema embebido, necesitamos crear un nuevo proyecto con EDK bajo Linux (Xilinx ISE+EDK 13.1 sobre Ubuntu 10.04). Una vez instalado este conjunto de herramientas, podemos volver a editar el fichero '.bashrc' como se explicó en el apartado anterior, y modificar la línea que cambiaba la variable de entorno "PATH" para dejarla tal y como podemos ver a continuación:

export PATH="RUTA_A_NUESTRO_REPOSITORIO"/microblaze-gnu/binaries/lin32-microblaze-unknown-linux-gnu_14.3_early/bin/:/opt/Xilinx/13.2/ISE_DS/ISE/bin/lin:$PATH

De esta forma, añadimos también al path la ruta a los binarios de las herramientas de Xilinx. Además de esto, para lanzar directamente las herramientas ISE y EDK, podemos crearnos unos scripts como los siguientes:

```
#!/bin/bash
source /opt/Xilinx/13.1/ISE_DS/settings32.sh
/opt/Xilinx/13.1/ISE_DS/EDK/bin/lin/xps
```

Tabla 1: Contenido del script para lanzar EDK

```
#!/bin/bash
source /opt/Xilinx/13.1/ISE_DS/settings32.sh
/opt/Xilinx/13.1/ISE_DS/ISE/bin/lin/ise
```

Tabla 2: Contenido del script para lanzar ISE

Debemos de tener en cuenta que el fichero de configuración que se está cargando (proporcionado por Xilinx, es el fichero 'settings32.sh'), es para la versión de 32 bits. Existe otro si trabajamos en un PC de 64 bits llamado 'settings64.sh'. Sin embargo, siempre que sea posible se recomienda el desarrollo con las herramientas de 32 bits, ya que se han detectado problemas en las versiones de 64 bits que pueden acarrearnos inconvenientes.

3.3.1 Consideraciones sobre el diseño en EDK

Dado que vamos a correr un Kernel compilado de GNU/Linux sobre Microblaze, existen ciertos elementos que debe tener nuestro sistema embebido como mínimo para poder ejecutarlo [DR6]:

- Una **MMU** (Memory-Management-Unit) en modo virtual y con dos zonas de protección de memoria.
- Un **timer** (xps_timer_core) con los dos timers internos activos.
- Una **UART** para la consola (UARTLite/UART 16550).
- **Controlador de interrupciones** con la UART y el timer conectados.

Estos requisitos mínimos son para un Kernel con soporte para memoria virtual (MMU). Xilinx no mantiene ninguna versión del Kernel para Microblaze que no posea MMU, aunque existen proyectos (como µC-Linux) que tienen versiones del Kernel sin soporte para unidad de manejo de memoria virtual. En este proyecto hemos optado por la versión soportada de Xilinx para MMU. Además, hemos agregado otros elementos Hardware a nuestro sistema embebido:

- Un core "mpmc" para tener la memoria DDR2 SDRAM de la placa disponible.
- Un core "xps_ethernetlite" para poder tener implementado el nivel MAC de Ethernet, de forma que nuestro diseño tenga conectividad a la red (imprescindible para acceder a la aplicación Web de forma remota).
- Un core "xps_iic" para acceder al bus de comunicaciones I2C.

En la siguiente imagen podemos ver un resumen de todos los elementos en el diseño EDK de nuestro System-on-Chip:

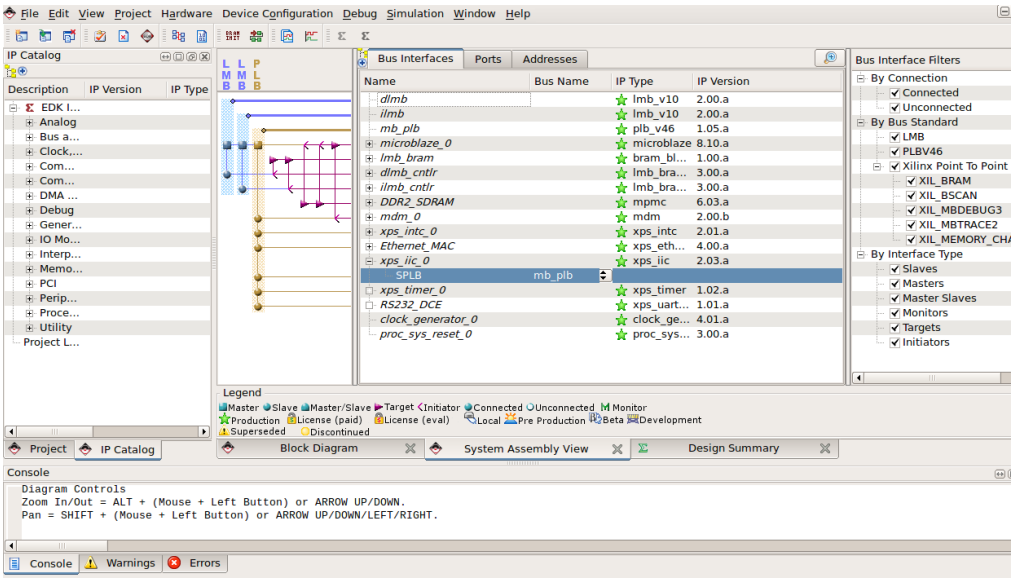


Ilustración 5: Interfaces del sistema embebido diseñado en EDK

Además, las dos salidas del bus I2C se configuran como puertos externos:

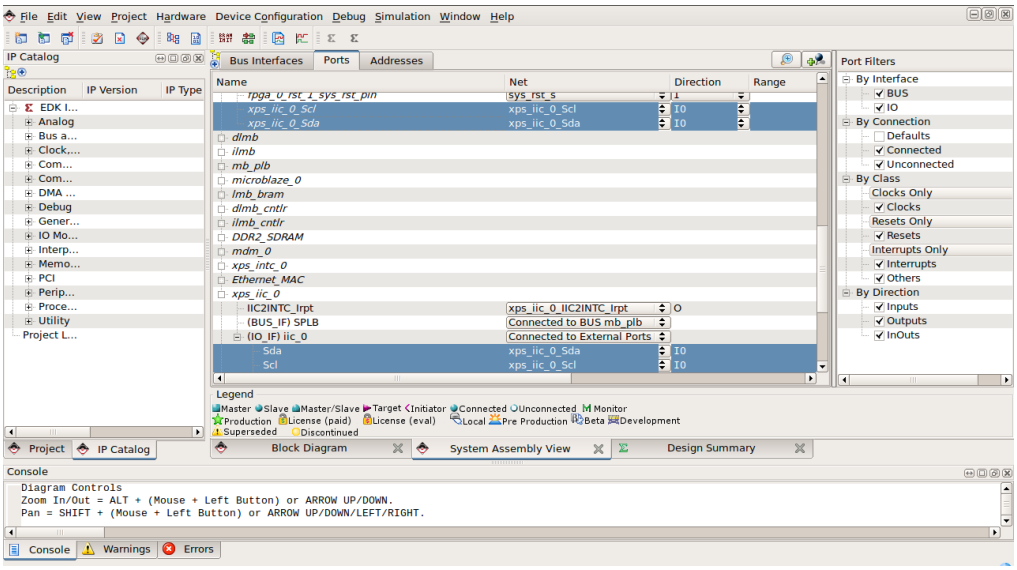


Ilustración 6: Puertos del bus I2C en el diseño EDK

Y estos puertos externos quedan fijados en el fichero de restricciones del diseño de la siguiente forma:

```
NET xps_iic_0_Scl LOC = A9  | IOSTANDARD=LVCMOS33;
NET xps_iic_0_Sda LOC = A12 | IOSTANDARD=LVCMOS33;
```

Tabla 3: Restricciones para el bus I2C en el fichero UCF

Tampoco debemos de olvidarnos de regenerar el mapa de direccionamiento al agregar el periférico I2C (y los que se añadan), y es importante tomar nota de la dirección base donde se encuentra nuestra memoria DDR2 SDRAM:

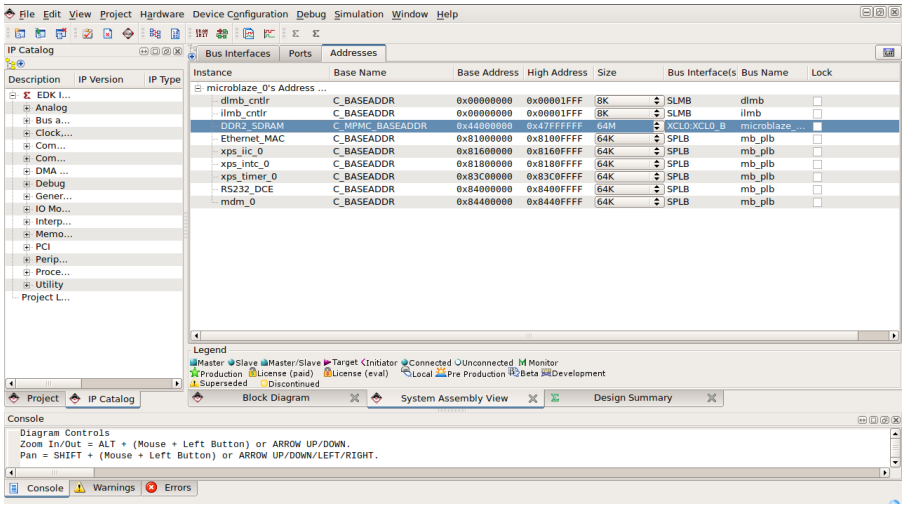


Ilustración 7: Mapa de direccionamiento del sistema embebido

El core que representa al procesador Microblaze debe ser configurado para habilitar algunos detalles significativos. Lo primero de todo, debemos de seleccionar la configuración 'Linux with MMU':

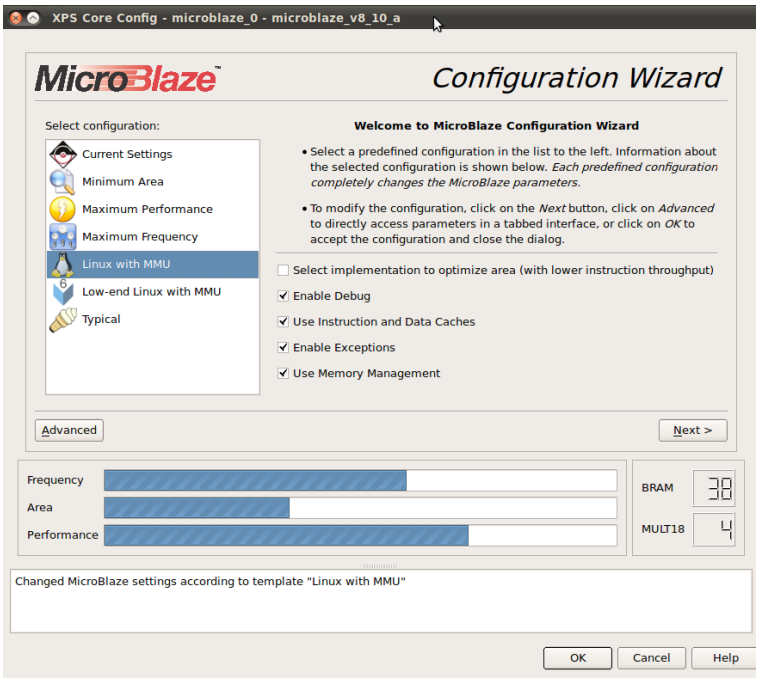


Ilustración 8: Selección de la configuración del procesador Microblaze

Si seguimos configurándolo, en el apartado 'General' habilitaremos el multiplicador de 32 bits (no vamos a emplear elementos de 64 bits, por lo que así ahorraremos algo de área):

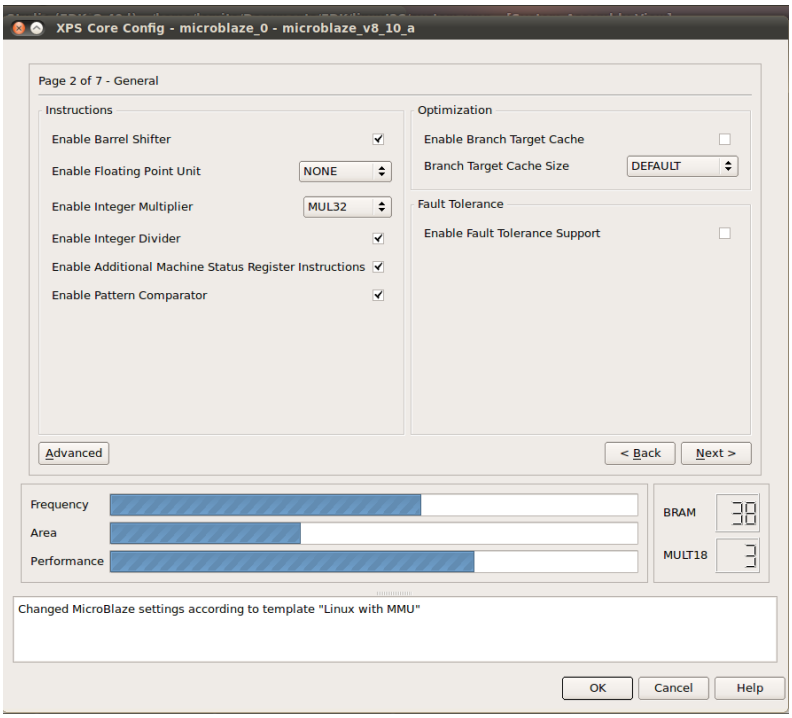


Ilustración 9: Configuración general del procesador Microblaze

En el quinto apartado, configuraremos las caches de instrucciones y de datos:

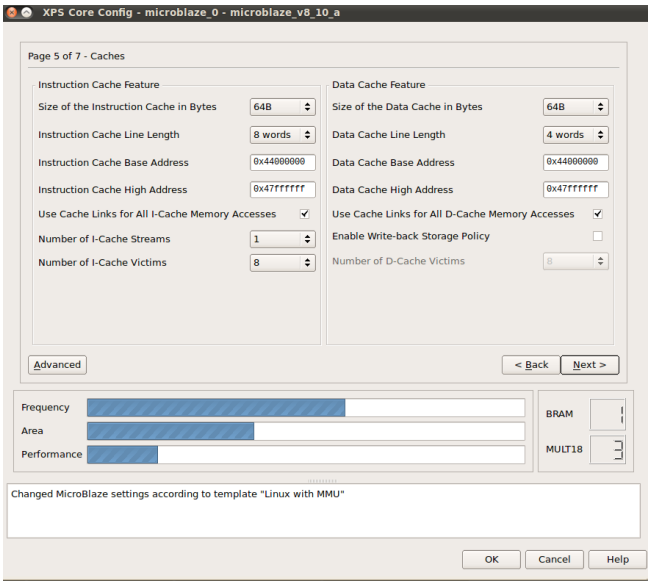


Ilustración 10: Configuración de las cachés en Microblaze

Como podemos ver, hemos configurado unas cachés muy pequeñas. Es importante habilitar estos elementos, pues el rendimiento que vamos a ganar muy remarcable frente a no habilitarlos. Sin embargo, seleccionamos tamaños de 64 bytes por dos motivos: el consumo de recursos en la FPGA, y que nuestras aplicaciones son muy sencillas y repiten los mismos procesos una y otra vez, por lo que no necesitamos grandes memorias de este tipo. También habilitaremos la configuración de la MMU, aunque los valores por defecto son correctos:

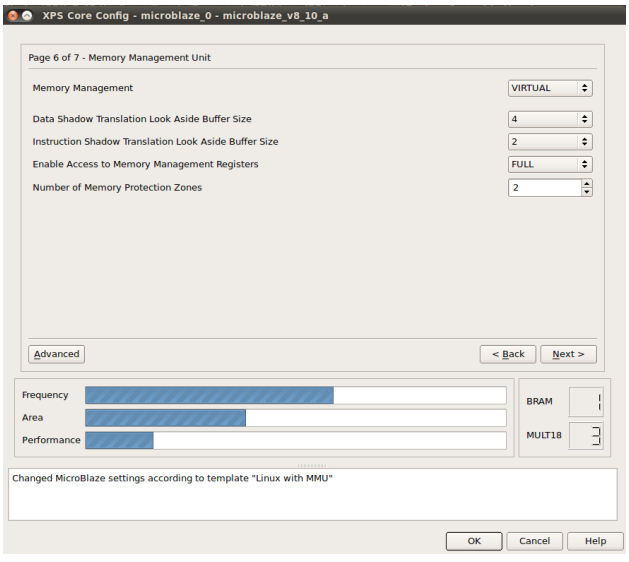


Ilustración 11: Configuración de la MMU en Microblaze

Con estos pasos, tenemos configurado nuestro sistema embebido para poder generar ejecutar Linux sobre Microblaze.

3.4 Compilando el Kernel de Linux

En este apartado, compilaremos el Kernel de Linux que nos descargamos desde los repositorios de Xilinx para generar la imagen que ejecutaremos sobre Microblaze. Los pasos a seguir son 3 básicamente:

1. Debemos generar el fichero DTS que contenga la descripción de nuestro Hardware.
2. Configuramos el núcleo del sistema operativo antes de compilarlo para modificar los parámetros necesarios.
3. Compilamos el Kernel para obtener el binario del sistema.

3.4.1 Generando el 'Device Tree' para nuestro diseño

Lo primero de todo, debemos de configurar EDK/XPS para que reconozca la herramienta 'Device tree generator'. Para ello, exportaremos nuestro diseño a XPS (la herramienta para desarrollar Software basada en Eclipse que nos ofrece Xilinx), y una vez hecho esto iremos a la opción de la barra de menús 'Xilinx Tools → Repositories'. En la ventana que se abre, en el apartado 'Global repositories', agregaremos una nueva entrada, y seleccionaremos el directorio 'device_tree_repo' donde nos clonamos la herramienta cuando preparamos el entorno de trabajo:

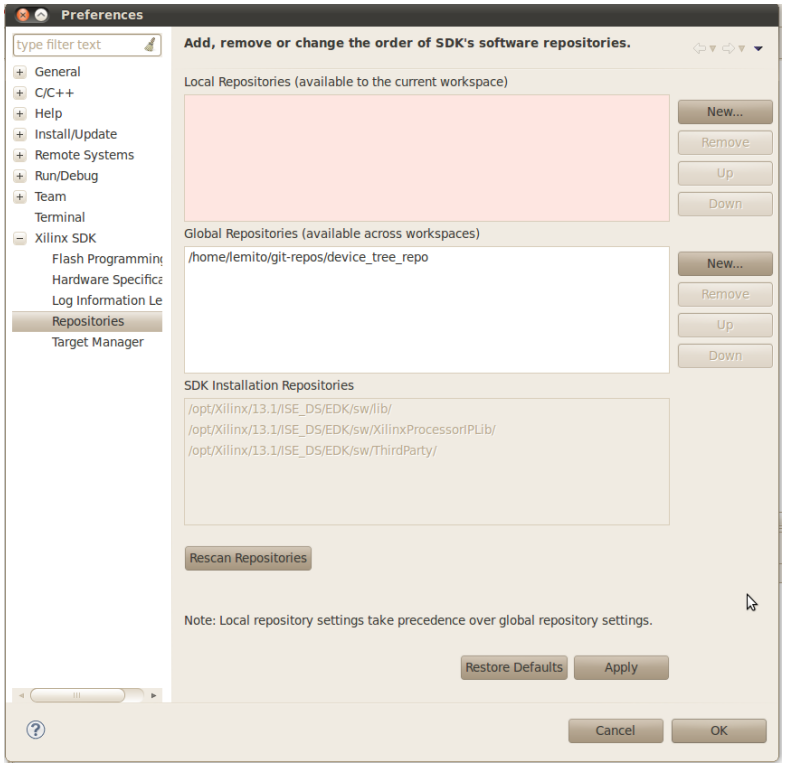


Ilustración 12: Configuración del repositorio 'device-tree' de Xilinx en XPS

Una vez hecho esto, pulsamos en 'Apply' y luego en 'OK'. Una vez hecho esto, iremos a la opción 'File → New → Xilinx Board Support Package' de la barra de menús. En la ventana que se nos abre, seleccionamos 'device-tree' en el apartado 'Board Support Package OS', y le damos el nombre que deseemos al proyecto. Pulsamos en 'Finish', y se nos abre un diálogo que nos permitirá configurar nuestro fichero DTS. En este diálogo aparecen 4 campos que podemos modificar:

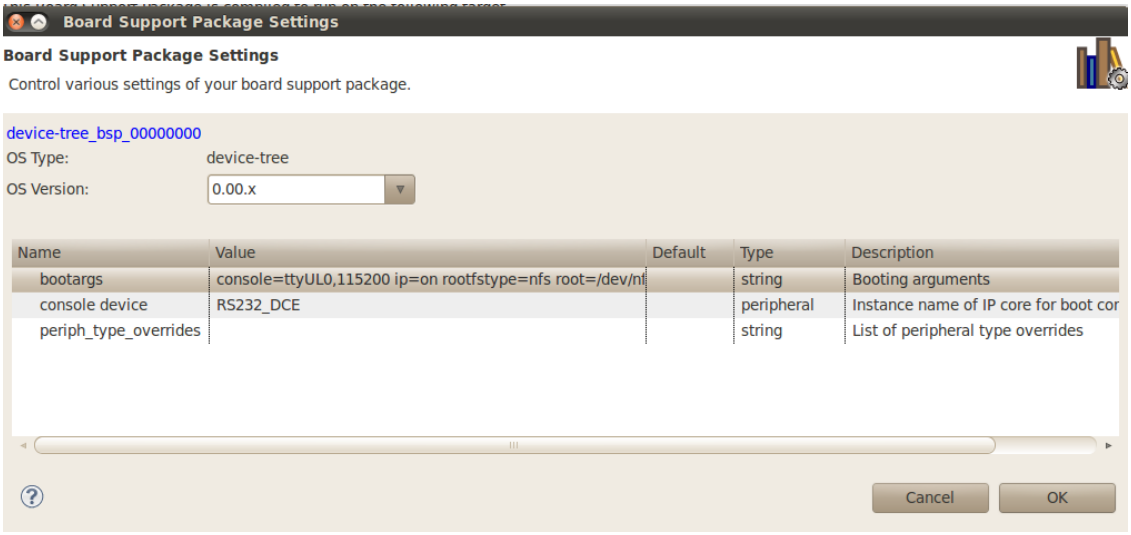


Ilustración 13: Configuración del BSP 'device-tree' en XPS

Estos 4 campos, como vemos, son:

- **OS version**: Versión del Sistema operativo. No se emplea en nuestro caso, así que podemos dejarlo como está.
- **Bootargs**: Este parámetro serán los argumentos que se le pasarán al Kernel durante el arranque del mismo. El valor que le daremos es "console=ttyUL0,115200 ip=on rootfstype=nfs root=/dev/nfs rw nfsroot=192.168.1.35:/shared/nfsroot2,tcp".
- **Console_device**: En este desplegable, seleccionaremos la UART que se empleará como consola del sistema (periférico HW).
- **Periph_type_overrides**: Lo dejamos en blanco.

Los argumentos de inicio que se le pasan al Kernel pueden variar dependiendo de la configuración que adoptemos. En nuestro caso, con el parámetro "console" le indicamos la consola del Kernel que empleará para asociarla al periférico HW seleccionado en "Console_device" junto con el baudrate al que configuramos el core UART de Xilinx. Con "ip=on" le decimos que use el servicio DHCP para obtener una dirección de red. En este punto, deberíamos asignar una dirección estática cuando se lleve a cabo un sistema final de producción. Esto se haría cambiando el valor de este parámetro por uno con el formato "ip=*<dirección-IP-estática>::<IP-puerta-de-enlace>:<máscara-de-red>:<nombre-del-sistema>:eth0:off*". Los siguientes parámetros "rootfstype", "root" y "nfsroot" indican al Kernel

que el tipo de sistema de ficheros para el directorio raíz del sistema de archivos que montará es “nfs” (Network File System), que lo monte en el directorio de su sistema de archivos “/dev/nfs rw” con permisos de lectura y escritura, y que acceda a él a través de la red al servidor NFS “192.168.1.35:/shared/nfsroot2, tcp” empleando el protocolo TCP. Se ha tomado esta decisión para el desarrollo del prototipo por dos motivos. La primera, resulta más cómodo a la hora de desarrollar montar el sistema de archivos por NFS (es decir, nuestro sistema de ficheros está en un servidor remoto, y accedemos a él a través de la red para montarlo en local y trabajar como si estuviéramos con un sistema de ficheros propio), ya que las modificaciones se hacen de forma instantánea y se reflejan tanto en el servidor como en el sistema embebido. Sin embargo, cuando se lleva un prototipo de estas características a producción, esta no es la aproximación más recomendable. En este caso, debemos de optar por dos posibilidades. Tener un disco RAM (ramdisk), o un sistema de ficheros en memoria FLASH. En nuestro caso, con las limitaciones de la placa de evaluación que tenemos no es viable ninguna de las otras dos opciones. Por una parte, tenemos muy poca memoria RAM, y cargar un ramdisk en ella ocuparía bastante espacio. Además, estos ficheros eliminan los cambios que se han realizado cuando se apaga el sistema (cada vez que iniciamos, el sistema de ficheros queda en el estado original cuando se generó esa imagen RAM del disco), cosa que no deseamos debido a los requisitos de nuestro entorno (para una puesta en producción, podría resultar idóneo). Otro inconveniente es el poco espacio que tienen las memorias flash en la placa de evaluación con la que estamos trabajando. Alojar en ellas el bitstream, el Kernel de Linux compilado (que ocupa unos 3.5 MB) y el sistema ramdisk resulta inviable (al pasar a producción, se puede montar una memoria flash de mayor tamaño, y alojar sin problema todos estos elementos para que se carguen en el arranque). El problema del poco tamaño de memoria no volátil también influye en la opción de alojar un sistema de ficheros flash. Estos sistemas de archivos emplean memoria no volátil para alojar el sistema de archivos, por lo que los cambios que realicemos serán permanentes y quedarán almacenados en el siguiente arranque aunque se apague el prototipo. Sin embargo, nos pasa lo mismo que con el disco RAM, no tenemos memoria suficiente para alojar todos estos componentes en la placa de evaluación. Por estos motivos, para el desarrollo realizado en este proyecto, se ha optado por montar el sistema de ficheros por NFS, tal y como se ha descrito anteriormente.

Volviendo a la generación del fichero DTS, una vez que hemos introducido los 4 parámetros en el diálogo, pulsamos en OK. Esto hará que se genere de forma automática el fichero DTS que deseamos con la configuración de nuestro sistema embebido. En el directorio de nuestro proyecto BSP, tenemos la ruta ‘microblaze_0 → libsrc → device_tree_v0_00_x’, dentro de la cuál podremos ver que existe el fichero ‘xilinx.dts’, que es el que describe nuestro HW. Si lo abrimos, tenemos un contenido parecido al que podemos ver en la siguiente tabla:

```
/dts-v1/;
/ {
	#address-cells = <1>;
	#size-cells = <1>;
	compatible = "xlnx,microblaze";
	model = "Xilinx MicroBlaze";
	DDR2_SDRAM: memory@44000000 {
		device_type = "memory";
		reg = < 0x44000000 0x4000000 >;
	} ;
```

```
	aliases {
		ethernet0 = &Ethernet_MAC;
		serial0 = &RS232_DCE;
	} ;
	chosen {
		bootargs = "console=ttyUL0,115200 ip=on rootfstype=nfs
root=/dev/nfs rw nfsroot=192.168.1.35:/shared/nfsroot2,tcp";
		linux,stdout-path = "/plb@0/serial@84000000";
	} ;
	cpus {
		#address-cells = <1>;
		#cpus = <0x1>;
		#size-cells = <0>;
		microblaze_0: cpu@0 {
			clock-frequency = <62500000>;
			compatible = "xlnx,microblaze-8.10.a";
			d-cache-baseaddr = <0x44000000>;
			d-cache-highaddr = <0x47ffffff>;
			d-cache-line-size = <0x10>;
			d-cache-size = <0x40>;
			device_type = "cpu";
			i-cache-baseaddr = <0x44000000>;
			i-cache-highaddr = <0x47ffffff>;
			i-cache-line-size = <0x20>;
			i-cache-size = <0x40>;
			model = "microblaze,8.10.a";
			reg = <0>;
			timebase-frequency = <62500000>;
			xlnx,addr-tag-bits = <0x14>;
			xlnx,allow-dcache-wr = <0x1>;
			xlnx,allow-icache-wr = <0x1>;
			xlnx,area-optimized = <0x0>;
			xlnx,branch-target-cache-size = <0x0>;
			xlnx,cache-byte-size = <0x40>;
			xlnx,d-axi = <0x0>;
			xlnx,d-lmb = <0x1>;
			xlnx,d-plb = <0x1>;
			xlnx,data-size = <0x20>;
			xlnx,dcache-addr-tag = <0x14>;
			xlnx,dcache-always-used = <0x1>;
			xlnx,dcache-byte-size = <0x40>;
			xlnx,dcache-data-width = <0x0>;
			xlnx,dcache-force-tag-lutram = <0x0>;
			xlnx,dcache-interface = <0x0>;
			xlnx,dcache-line-len = <0x4>;
			xlnx,dcache-use-fsl = <0x1>;
			xlnx,dcache-use-writeback = <0x0>;
			xlnx,dcache-victims = <0x0>;
			xlnx,debug-enabled = <0x1>;
			xlnx,div-zero-exception = <0x1>;
			xlnx,dynamic-bus-sizing = <0x1>;
			xlnx,ecc-use-ce-exception = <0x0>;
			xlnx,edge-is-positive = <0x1>;
			xlnx,endianness = <0x0>;
			xlnx,family = "spartan3a";
			xlnx,fault-tolerant = <0x0>;
			xlnx,fpu-exception = <0x0>;
			xlnx,freq = <0x3b9aca0>;
			xlnx,fsl-data-size = <0x20>;
```

```
                    xlnx,fsl-exception = <0x0>;
                    xlnx,fsl-links = <0x0>;
                    xlnx,i-axi = <0x0>;
                    xlnx,i-lmb = <0x1>;
                    xlnx,i-plb = <0x1>;
                    xlnx,icache-always-used = <0x1>;
                    xlnx,icache-data-width = <0x0>;
                    xlnx,icache-force-tag-lutram = <0x0>;
                    xlnx,icache-interface = <0x0>;
                    xlnx,icache-line-len = <0x8>;
                    xlnx,icache-streams = <0x1>;
                    xlnx,icache-use-fsl = <0x1>;
                    xlnx,icache-victims = <0x8>;
                    xlnx,ill-opcode-exception = <0x1>;
                    xlnx,instance = "microblaze_0";
                    xlnx,interconnect = <0x1>;
                    xlnx,interrupt-is-edge = <0x0>;
                    xlnx,mmu-dtlb-size = <0x4>;
                    xlnx,mmu-itlb-size = <0x2>;
                    xlnx,mmu-privileged-instr = <0x0>;
                    xlnx,mmu-tlb-access = <0x3>;
                    xlnx,mmu-zones = <0x2>;
                    xlnx,number-of-pc-brk = <0x1>;
                    xlnx,number-of-rd-addr-brk = <0x0>;
                    xlnx,number-of-wr-addr-brk = <0x0>;
                    xlnx,opcode-0x0-illegal = <0x1>;
                    xlnx,optimization = <0x0>;
                    xlnx,pvr = <0x2>;
                    xlnx,pvr-user1 = <0x0>;
                    xlnx,pvr-user2 = <0x0>;
                    xlnx,reset-msr = <0x0>;
                    xlnx,sco = <0x0>;
                    xlnx,stream-interconnect = <0x0>;
                    xlnx,unaligned-exceptions = <0x1>;
                    xlnx,use-barrel = <0x1>;
                    xlnx,use-branch-target-cache = <0x0>;
                    xlnx,use-dcache = <0x1>;
                    xlnx,use-div = <0x1>;
                    xlnx,use-ext-brk = <0x1>;
                    xlnx,use-ext-nm-brk = <0x1>;
                    xlnx,use-extended-fsl-instr = <0x0>;
                    xlnx,use-fpu = <0x0>;
                    xlnx,use-hw-mul = <0x1>;
                    xlnx,use-icache = <0x1>;
                    xlnx,use-interrupt = <0x1>;
                    xlnx,use-mmu = <0x3>;
                    xlnx,use-msr-instr = <0x1>;
                    xlnx,use-pcmp-instr = <0x1>;
                    xlnx,use-stack-protection = <0x0>;
              } ;
      } ;
      mb_plb: plb@0 {
              #address-cells = <1>;
              #size-cells = <1>;
              compatible   =   "xlnx,plb-v46-1.05.a",   "xlnx,plb-v46-1.00.a",
"simple-bus";
              ranges ;
              Ethernet_MAC: ethernet@81000000 {
                    compatible  =  "xlnx,xps-ethernetlite-4.00.a",  "xlnx,xps-
```

```
ethernetlite-1.00.a";
                    device_type = "network";
                    interrupt-parent = <&xps_intc_0>;
                    interrupts = < 2 0 >;
                    local-mac-address = [ 00 0a 35 00 00 00 ];
                    reg = < 0x81000000 0x10000 >;
                    xlnx,duplex = <0x1>;
                    xlnx,family = "spartan3a";
                    xlnx,include-global-buffers = <0x0>;
                    xlnx,include-internal-loopback = <0x0>;
                    xlnx,include-mdio = <0x1>;
                    xlnx,rx-ping-pong = <0x0>;
                    xlnx,tx-ping-pong = <0x0>;
              } ;
              RS232_DCE: serial@84000000 {
                    clock-frequency = <62500000>;
                    compatible   =   "xlnx,xps-uartlite-1.01.a",   "xlnx,xps-
uartlite-1.00.a";
                    current-speed = <115200>;
                    device_type = "serial";
                    interrupt-parent = <&xps_intc_0>;
                    interrupts = < 3 0 >;
                    port-number = <0>;
                    reg = < 0x84000000 0x10000 >;
                    xlnx,baudrate = <0x1c200>;
                    xlnx,data-bits = <0x8>;
                    xlnx,family = "spartan3a";
                    xlnx,odd-parity = <0x0>;
                    xlnx,use-parity = <0x0>;
              } ;
              mdm_0: serial@84400000 {
                    compatible   =   "xlnx,mdm-2.00.b",   "xlnx,xps-uartlite-
1.00.a";
                    reg = < 0x84400000 0x10000 >;
                    xlnx,family = "spartan3a";
                    xlnx,interconnect = <0x1>;
                    xlnx,jtag-chain = <0x2>;
                    xlnx,mb-dbg-ports = <0x1>;
                    xlnx,use-uart = <0x1>;
              } ;
              xps_iic_0: i2c@81600000 {
                    compatible   =   "xlnx,xps-iic-2.03.a",   "xlnx,xps-iic-
2.00.a";
                    interrupt-parent = <&xps_intc_0>;
                    interrupts = < 0 2 >;
                    reg = < 0x81600000 0x10000 >;
                    xlnx,clk-freq = <0x3b9aca0>;
                    xlnx,family = "spartan3a";
                    xlnx,gpo-width = <0x1>;
                    xlnx,iic-freq = <0x186a0>;
                    xlnx,scl-inertial-delay = <0x5>;
                    xlnx,sda-inertial-delay = <0x5>;
                    xlnx,ten-bit-adr = <0x0>;
              } ;
              xps_intc_0: interrupt-controller@81800000 {
                    #interrupt-cells = <0x2>;
                    compatible   =   "xlnx,xps-intc-2.01.a",   "xlnx,xps-intc-
1.00.a";
                    interrupt-controller ;
```

```
                reg = < 0x81800000 0x10000 >;
                xlnx,kind-of-intr = <0xe>;
                xlnx,num-intr-inputs = <0x4>;
            } ;
            xps_timer_0: timer@83c00000 {
                compatible  =  "xlnx,xps-timer-1.02.a",  "xlnx,xps-timer-
1.00.a";
                interrupt-parent = <&xps_intc_0>;
                interrupts = < 1 0 >;
                reg = < 0x83c00000 0x10000 >;
                xlnx,count-width = <0x20>;
                xlnx,family = "spartan3a";
                xlnx,gen0-assert = <0x1>;
                xlnx,gen1-assert = <0x1>;
                xlnx,one-timer-only = <0x0>;
                xlnx,trig0-assert = <0x1>;
                xlnx,trig1-assert = <0x1>;
            } ;
        } ;
} ;
```

Tabla 4: Ejemplo de fichero DTS

Como vemos, este fichero describe todo el HW que tenemos configurado en nuestro sistema embebido. Por ejemplo, para el periférico I2C que agregamos al diseño, tenemos la entrada "xps_iic_0" que indica la dirección de la memoria donde está mapeado "81600000", y los registros de parametrización del periférico ("compatible" indica los drivers que se pueden cargar para manejarlo, "interrupt-parent" cuál es el controlador de interrupciones al que se conecta, "interrupts" la máscara identificadora asignada a la interrupción que se le asigna, etc.). También una serie de argumentos "xlnx, *XXXX*" que se cargan en el sistema operativo, y que serán accedidos por el driver para tener información sobre cómo se encuentra configurado el periférico HW y poder así manejarlo correctamente.

En definitiva, este fichero contiene toda la descripción HW de nuestro sistema embebido que el Kernel necesita durante su etapa de arranque. Para que se compile correctamente, debemos de copiarlo al siguiente directorio (en el repositorio donde nos descargamos el código del Kernel) "<path_al_repositorio>/arch/microblaze/boot/dts/". Esta es la ubicación en la que el sistema de construcción del Kernel buscará el DTS para convertirlo a su correspondiente DTB.

3.4.2 Configurando el Kernel de GNU/Linux

El siguiente paso es configurar el Kernel de GNU/Linux. Durante este proceso, podremos parametrizar el núcleo de este Sistema Operativo y seleccionar los módulos y funcionalidades que deseamos que tenga. De esta forma, podemos eliminar aspectos del sistema que no nos interesan (por ejemplo, la conectividad Wifi o Bluetooth, que para nosotros no tiene ningún sentido). Tras este paso compilaremos el núcleo con las opciones seleccionadas, y obtendremos una imagen del sistema a medida para nuestras necesidades. A

menor número de elementos que incluyamos en el Kernel final mejor, ya que obtenemos ciertas ventajas: por una parte, el tamaño que ocupará la imagen final, mucho menor y por lo tanto tendremos unos requisitos menos restrictivos en este aspecto para su ejecución. Por otro lado nuestro sistema será mucho más estable, ya que cuantos menos componentes carguemos en el mismo, menores son las probabilidades de que alguno de ellos dalle por errores en el código de los mismos.

El proceso de configuración se basa en generar un fichero '.config' que contiene los parámetros que se pasarán al sistema de construcción del núcleo para agregar los componentes seleccionados. Por suerte, tenemos herramientas más visuales que nos ayudan a generar este fichero para no tener que escribirlo a mano, lo cual sería un proceso lento, tedioso y propenso a fallos.

Lo primero que debemos hacer es posicionarnos en el directorio raíz del repositorio GIT con el código fuente del Kernel. Todas las rutas que indiquemos a partir de ahora serán tomando esta carpeta como raíz. Para crear nuestra configuración nos ayudaremos de unos ficheros básicos que nos proporciona Xilinx para la arquitectura Microblaze. El que nos interesa a nosotros se encuentra en 'arch/Microblaze/configs/mmu_defconfig'. Este archivo es una configuración del núcleo por defecto para un procesador Microblaze con MMU y sistema de ficheros *ramdisk* sobre la placa de evaluación 'Spartan-6 SP605' de Xilinx. Por lo tanto, debemos de copiarnos este archivo a la raíz de nuestro repositorio y renombrarlo, lo cual podemos hacer con el siguiente comando:

$> cp arch/microblaze/configs/mmu_defconfig ./.config

Ahora, arrancaremos el sistema de configuración del núcleo. Para ello, debemos emplear el siguiente comando:

$> make ARCH=microblaze xconfig

Esto arrancará una aplicación de escritorio que nos permitirá parametrizar nuestro núcleo, tomando como base la configuración que figura en el archivo '.config' que acabamos de copiarnos (existen más herramientas, como una basada en consola que podemos invocar con 'menuconfig' en vez de 'xconfig'). Esta pantalla general la podemos ver en la siguiente imagen:

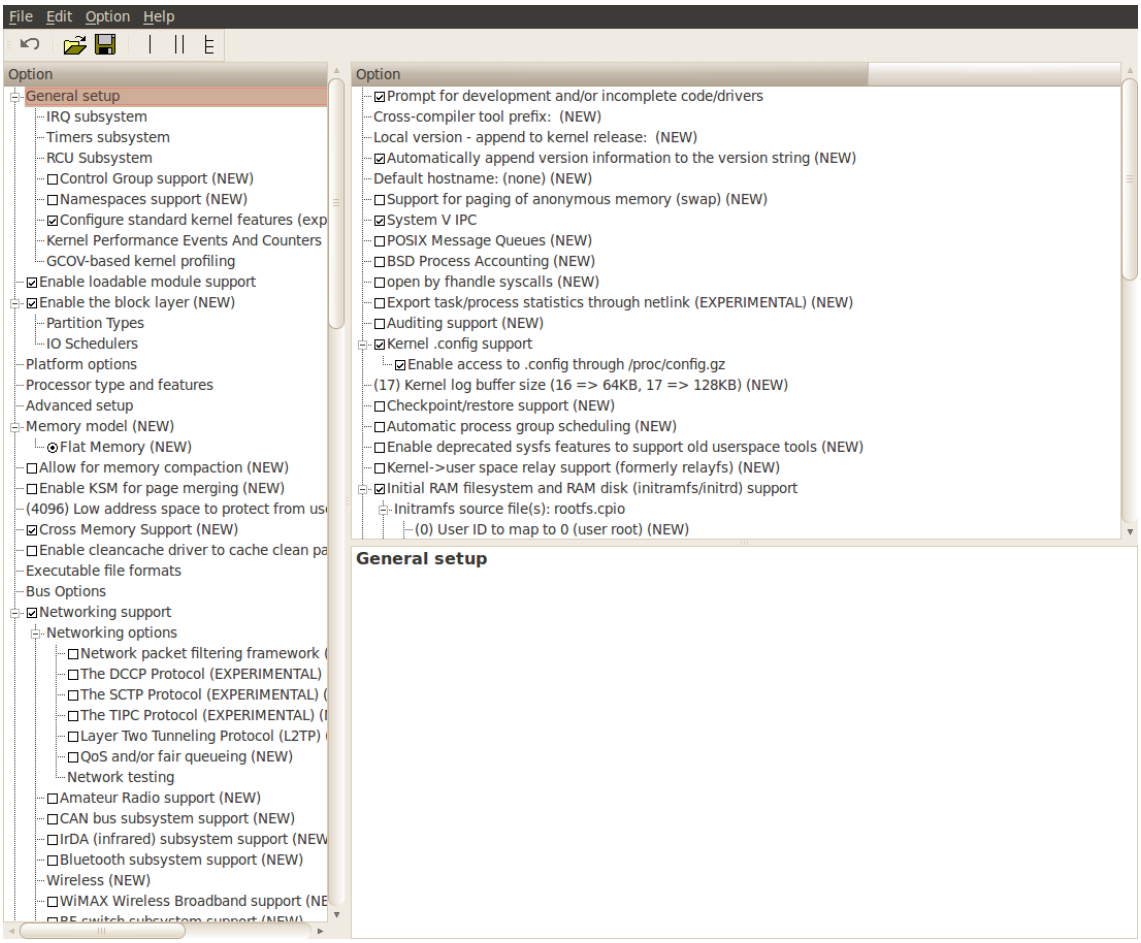


Ilustración 14: Pantalla principal de la aplicación para configurar el Kernel

Las opciones que vamos a configurar son:

1. **General setup → Initial RAM filesystem and RAM disk (initramfs/initrd) support**: Lo deshabilitamos, ya que nuestro sistema no va a basarse en un disco RAM.
2. **Platform options → Physical Address where Linux Kernel is**: En esta opción, debemos de indicar la dirección base de nuestro mapa de direccionamiento para el periférico de la memoria DDR2 SDRAM. En el caso del DTS de ejemplo, sería "0x44000000"
3. **Platform options →Targeted FPGA family:** 'spartan3a'.
4. **Platform options →USE_MSR_INSTR:** Ponemos el mismo valor que figure en nuestro DTS, en el parámetro "xlnx,use-msr-instr", en este caso '1'.
5. **Platform options →USE_PCMP_INSTR:** Valor del parámetro 'xlnx,use-pcmp-instr' en el DTS.
6. **Platform options →USE_BARREL:** Valor del parámetro 'xlnx,use-barrel' en el DTS.
7. **Platform options →USE_DIV:** Valor del parámetro 'xlnx,use-div' en el DTS.
8. **Platform options →USE_HW_MUL:** Valor del parámetro 'xlnx,use-hw-mul' en el DTS.
9. **Platform options →USE_FPU:** Valor del parámetro 'xlnx,use-fpu' en el DTS.

10. **Platform options → Core version number:** En este parámetro pondremos la versión del procesador Microblaze que empleemos. En nuestro caso es '8.10.a'.
11. **Processor type and features → Default bootloader Kernel arguments → Default Kernel command String:** Aquí debemos de poner el valor que le dimos a la herramienta de Xilinx para generar el DTS, en nuestro caso '*console=ttyUL0,115200 ip=on rootfstype=nfs root=/dev/nfs rw nfsroot=192.168.1.35:/shared/nfsroot2,tcp*'.
12. **Networking support → Networking options → TCP/IP networking → IP: Kernel level autoconfiguration:** Marcamos esta opción, lo que hará que podamos seleccionar otras opciones que dependen de ella.
13. **Networking support → Networking options → TCP/IP networking → IP: Kernel level autoconfiguration → IP: DHCP support:** Marcamos esta opción. Es necesaria ya que vamos a emplear DHCP para obtener nuestra configuración de red.
14. **Device Drivers → I2C support:** Marcamos esta opción como parte del Kernel (un 'tick' en el checkbox), no como un módulo que pueda ser cargado (que sería un 'círculo' en el checkbox). Al marcarla, podremos seleccionar una serie de elementos que nos interesan y que dependen de ella.
15. **Device Drivers → I2C support →Autoselect pertinent helper modules:** Desmarcamos esta opción.
16. **Device Drivers → I2C support →I2C device interface:** Marcamos esta opción para que la compile junto al Kernel.
17. **Device Drivers → I2C support →I2C algorithms → Xilinx IIC interface:** La marcamos para que forme parte del sistema operativo.
18. **Device Drivers → I2C support → I2C Hardware Bus support → Xilinx I2C controller:** Lo marcamos para que forme parte del Kernel.
19. **File Systems → Network File Systems → NFS Client Support → NFS client support for NFS version 3 → NFS client support for the NFSv3 ACL protocol extension:** Lo marcamos para que forme parte del Kernel del sistema.
20. **File Systems → Network File Systems → NFS Client Support → NFS client support for NFS version 4:** Lo marcamos para que forme parte del Kernel.
21. **File Systems → Network File Systems → Root file system on NFS:** Lo marcamos para que forme parte del Kernel.
22. **File Systems → Network File Systems → CIFS support (Advanced network filesystem, SMBFS successor):** Lo desmarcamos.

Con estas opciones seleccionadas, podemos guardar la configuración realizada ("File → save") y salir de la herramienta ("File → Quit"). El archivo .config ya ha sido generado.

3.4.3 Compilando el Kernel de GNU/Linux

Antes de proceder a la compilación del Kernel, debemos de hacer una pequeña modificación al archivo Makefile que emplearemos para ello. Se encuentra ubicado en "arch/microblaze/boot/Makefile", y debemos buscar y eliminar la siguiente línea:

```
$(obj)/simpleImage.%: vmlinux FORCE
    $(call if_changed,cp,.unstrip)
    $(call if_changed,objcopy)
    $(call if_changed,uimage)
    $(call if_changed,strip)
@echo 'Kernel: $@ is ready' ' (#'`cat .version`')'
```

Tabla 5: Modificación del fichero Makefile para compilar el Kernel de GNU/Linux

Esta modificación es necesaria ya que si no el sistema de construcción siempre intentará generar una imagen U-Boot, y nos dará un mensaje de error. Un cargador de arranque (como U-Boot), básicamente, es un elemento Software que se ejecuta al arrancar nuestro sistema embebido. Entre las funciones mínimas que realizaría, estaría la inicialización del hardware, el paso de parámetros de arranque al sistema operativo, y el inicio del sistema operativo, cediéndole el control al mismo para que arranque. Típicamente, en un sistema embebido se tiene en la memoria el bootloader, a continuación el Kernel del sistema, y seguido el sistema de ficheros. De esta manera, nada más arrancar el sistema, se ejecutaría el bootloader que realizaría las operaciones necesarias y cedería el control al sistema operativo, quien estaría configurado para montar el sistema de ficheros ubicado en la propia memoria. Sin embargo, en nuestro caso no vamos a emplear un cargador de arranque, ya que hemos configurado el Kernel para que compruebe el Hardware por sí mismo (con el fichero DTS) y monte el sistema de archivos a través de NFS. Si generásemos una imagen con bootloader, básicamente lo que haríamos sería compilar el DTS junto al bootloader y al Kernel, y seguir la secuencia descrita anteriormente.

Con esta modificación al fichero Makefile, ya estamos en disposición de lanzar la compilación del Kernel. Para ello, nos posicionaremos en la raíz del repositorio con el código fuente, y ejecutaremos el siguiente comando (si no hemos agregado al fichero '.bashrc' la entrada "export CROSS_COMPILE=…", deberíamos de añadir este parámetro en el comando de compilación):

$> make -j4 ARCH=microblaze simpleImage.xilinx

El modificador '-j4' le indica al sistema de construcción que tenemos 4 núcleos en nuestra máquina, por lo que podrá realizar una compilación paralela y así ahorrarnos tiempo. Tras esperar unos minutos, debemos ver un mensaje final parecido al siguiente:

```
Kernel: arch/Microblaze/boot/simpleImage.xilinx is ready (#2)
```

Tabla 6: Mensaje indicativo de una compilación del Kernel satisfactoria

Si todo el proceso se llevó a cabo de manera satisfactoria, ya tendremos nuestra imagen lista para descargar en la FPGA junto al bitstream, y comenzar a emplear nuestro sistema Linux. El binario se encuentra en 'arch/microblaze/boot/simpleImage.xilinx'.

3.5 Creando el sistema de ficheros a exportar por NFS

El siguiente paso que debemos llevar a cabo es crear el sistema de archivos que vamos a exportar por NFS para que nuestro sistema embebido lo monte a través de la red y trabaje con él. El proceso de generación de un sistema de ficheros puede ser muy tedioso si se lleva a cabo de forma manual (para más detalles sobre este proceso, consultar [DR15]), por lo que vamos a partir de un disco RAM comprimido que nos ofrece Xilinx. Para ello, creamos un directorio sobre el que trabajar, y nos posicionamos en el mismo. Para descargar esta imagen que ya viene preparada con 'busybox', ejecutamos el siguiente comando:

$>wget http://www.wiki.xilinx.com/file/view/microblaze_complete.cpio.gz/419243588/microblaze_complete.cpio.gz

Con esto, descargamos la imagen que nos ofrece Xilinx. Para descomprimirla y crear nuestro sistema de ficheros raíz que empleará el Sistema Operativo para funcionar, primero creamos otro directorio (que será el que exportaremos por NFS y el que nuestro Sistema Operativo embebido considerará "/"), nos posicionamos en él, y ejecutamos el siguiente comando (como usuario *root*):

$> gunzip –c <path_al_ramdisk>/microblaze_complete.cpio.gz | cpio –i

Si listamos el contenido de este directorio, deberíamos ver una serie de carpetas como en la siguiente imagen:

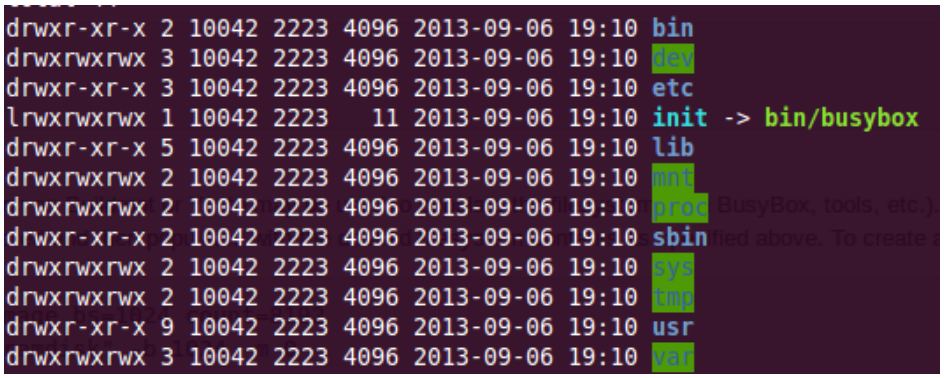


Ilustración 15: Contenido del directorio 'rootfs' que se exportará por NFS

Tras este proceso, ya tenemos listo nuestro sistema de ficheros raíz que será montado a través de NFS. Lo siguiente que debemos tener configurado es un servidor NFS en nuestra máquina para que el sistema embebido pueda montarlo de forma remota. Sin embargo, no es objeto de este proyecto explicar este proceso, para lo que podemos consultar por ejemplo los tutoriales [DR30] o [DR31].

3.6 Programando la FPGA y descargando el Kernel para su ejecución

Para automatizar el proceso de programación de la FPGA y descarga del sistema Linux para su ejecución, podemos emplear un Script como el siguiente:

```
#!/bin/bash
source /opt/Xilinx/13.1/ISE_DS/settings32.sh

cd <path_al_proyecto_EDK_del_SoC>

impact -batch etc/download.cmd

xterm -e xmd -opt etc/xmd_microblaze_0.opt&

gksudo -S -D GKTERM "gtkterm -p /dev/ttyS0 -s 115200&"
```

Tabla 7: Script para la programación de la FPGA y la descarga y ejecución del Kernel

Previamente, debemos haber copiado la imagen 'simpleImage.xilinx' a nuestro directorio con el proyecto EDK del sistema embebido. También tenemos que crearnos el fichero 'xmd_microblaze_0.opt' dentro del directorio 'opt' del proyecto EDK. Este fichero tendrá el siguiente contenido:

```
connect mb mdm –debugdevice cpunr 1
dow simpleImage.xilinx
run
```

Tabla 8: Fichero para la descarga y ejecución del Kernel

Además, también debemos instalar el programa GTKTERM, u otro que nos permita conectarnos a un terminal serie, y configurar el Script para que lo lance automáticamente. Con estos Script, lo que conseguimos es que la placa se programe con el bitstream lo primero, y luego conecte mediante XMB al procesador Microblaze, descargue el Kernel a la memoria (este proceso es lento, tarda unos minutos) y lo ejecute, mientras conecta el programa GTKTERM al puerto serie para poder comunicarnos con el Sistema Operativo. Al final, lo que veremos será el log de arranque del sistema, y una consola de comandos del sistema en el puerto serie. Un log de arranque típico del sistema lo podemos ver a continuación:

```
Early console on uartlite at 0x84000000
bootconsole [earlyser0] enabled
Ramdisk addr 0x0000003f, Compiled-in FDT at 0xc0262a70
Linux version 3.6.0-dirty (lemito@lemito-desktop-ubuntu) (gcc version 4.6.2
20111018 (prerelease) (crosstool-NG 1.14.1) ) #5 Fri Sep 6 21:54:11 CEST
2013
setup_cpuinfo: initialising
setup_cpuinfo: Using full CPU PVR support
cache: wt_msr_noirq
setup_memory: max_mapnr: 0x4000
setup_memory: min_low_pfn: 0x44000
setup_memory: max_low_pfn: 0x48000
setup_memory: max_pfn: 0x48000
Zone ranges:
  DMA      [mem 0x44000000-0x47ffffff]
  Normal   empty
Movable zone start for each node
Early memory node ranges
  node   0: [mem 0x44000000-0x47ffffff]
On node 0 totalpages: 16384
free_area_init_node: node 0, pgdat c031c3c0, node_mem_map c0367000
  DMA zone: 128 pages used for memmap
  DMA zone: 0 pages reserved
  DMA zone: 16256 pages, LIFO batch:3
early_printk_console remapping from 0x84000000 to 0xffffd000
pcpu-alloc: s0 r0 d32768 u32768 alloc=1*32768
pcpu-alloc: [0] 0
Built 1 zonelists in Zone order, mobility grouping on.  Total pages: 16256
Kernel    command    line:    console=ttyUL0,115200    ip=on    rootfstype=nfs
root=/dev/nfs rw nfsroot=192.168.1.35:/shared/nfsroot2,tcp
PID hash table entries: 256 (order: -2, 1024 bytes)
Dentry cache hash table entries: 8192 (order: 3, 32768 bytes)
Inode-cache hash table entries: 4096 (order: 2, 16384 bytes)
Memory: 61380k/65536k available (3080k kernel code, 4156k reserved, 107k
data, 168k bss, 114k init)
Kernel virtual memory layout:
  * 0xfffe000..0xffff000  : fixmap
  * 0xffffd000..0xfffffe000  : early ioremap
  * 0xf0000000..0xffffd000  : vmalloc & ioremap
NR_IRQS:33
interrupt-controller #0 at 0xf0000000, num_irq=4, edge=0xe
No chosen timer found, using default
timer #0 at 0xf0002000, irq=1
microblaze_timer_set_mode: shutdown
microblaze_timer_set_mode: periodic
Calibrating delay loop... 30.10 BogoMIPS (lpj=150528)
pid_max: default: 4096 minimum: 301
Mount-cache hash table entries: 512
NET: Registered protocol family 16
bio: create slab <bio-0> at 0
Switching to clocksource microblaze_clocksource
NET: Registered protocol family 2
TCP established hash table entries: 2048 (order: 2, 16384 bytes)
TCP bind hash table entries: 2048 (order: 3, 40960 bytes)
TCP: Hash tables configured (established 2048 bind 2048)
TCP: reno registered
UDP hash table entries: 128 (order: 0, 6144 bytes)
UDP-Lite hash table entries: 128 (order: 0, 6144 bytes)
NET: Registered protocol family 1
```

```
RPC: Registered named UNIX socket transport module.
RPC: Registered udp transport module.
RPC: Registered tcp transport module.
RPC: Registered tcp NFSv4.1 backchannel transport module.
NFS: Registering the id_resolver key type
Key type id_resolver registered
Key type id_legacy registered
msgmni has been set to 119
io scheduler noop registered
io scheduler deadline registered
io scheduler cfq registered (default)
84000000.serial: ttyUL0 at MMIO 0x84000000 (irq = 3) is a uartlite
console [ttyUL0] enabled, bootconsole disabled
console [ttyUL0] enabled, bootconsole disabled
uartlite 84400000.serial: failed to get alias id, errno -19
uartlite 84400000.serial: failed to get port-number
brd: module loaded
xilinx_emaclite 81000000.ethernet: Device Tree Probing
No mdio handler
xilinx_emaclite 81000000.ethernet: error registering MDIO bus
xilinx_emaclite 81000000.ethernet: MAC address is now 00:0a:35:00:00:00
xilinx_emaclite 81000000.ethernet: Xilinx EmacLite at 0x81000000 mapped to
0xF0020000, irq=2
i2c /dev entries driver
Device Tree Probing 'i2c'
xilinx-iic #0 at 0x81600000 mapped to 0xF0040000, irq=4
TCP: cubic registered
NET: Registered protocol family 17
Key type dns_resolver registered
Sending DHCP requests ..., OK
IP-Config: Got DHCP answer from 192.168.1.1, my address is 192.168.1.36
IP-Config: Complete:
     device=eth0, addr=192.168.1.36, mask=255.255.255.0, gw=192.168.1.1
     host=dhcppc3, domain=, nis-domain=(none)
     bootserver=192.168.1.1, rootserver=192.168.1.35, rootpath=
VFS: Mounted root (nfs filesystem) on device 0:9.
Freeing unused kernel memory: 114k freed
Starting rcS...
++ Creating device points
mkdir: cannot create directory '/dev/pts': File exists
++ Mounting filesystem
++ Loading system loggers
++ Starting telnet daemon
rcS Complete
/bin/sh: can't access tty; job control turned off
/ #
```

Tabla 9: Log de ejemplo de arranque del sistema

3.7 Configuración del usuario root

Nuestro sistema viene sin información sobre grupos o usuarios. Por defecto, el usuario con el que nos conectamos es 'root'. Sin embargo, al no tener la citada información que identifique a los usuarios y sus grupos, el Sistema Operativo no es capaz de manejar de manera adecuada el sistema de ficheros, máxime si pensamos en que lo estamos exportando por NFS y

que cuando hacemos modificaciones desde el servidor NFS estas quedan reflejadas (usuario/grupo propietario, etc). Por lo tanto, para que nuestro usuario (con el que se ejecutará nuestro SW embebido) no tenga problemas de permisos a la hora de manejar el sistema de ficheros, debe ser dado de alta. Este proceso es muy sencillo, pues para ello se emplean únicamente dos ficheros: 'group' y 'passwd'. El primero define los grupos existentes en el sistema, y el segundo los usuarios junto a sus contraseñas (aunque insertar las contraseñas en este fichero ya es una práctica en desuso desde hace mucho. El campo correspondiente suele tener valor 'x', y se emplea el fichero 'shadow' que contiene el HASH de las contraseñas, de forma que no se encuentren visibles en texto plano). De todas formas, nuestra configuración va a ser la más básica, pues únicamente daremos de alta al usuario administrador 'root' sin ningún password, y al grupo 'root' al que pertenece. Para ello, crearemos el fichero /etc/passwd' con el siguiente contenido (suponemos en este apartado del documento que estamos trabajando ya dentro de nuestro sistema embebido):

```
root::0:0:root:/root:/bin/sh
```

Tabla 10: Contenido del fichero 'passwd'

Esta entrada en el fichero contiene una serie de campos. El primero es el nombre del usuario, 'root'. El segundo, que hemos dejado en blanco, sería la contraseña, pero nosotros lo dejamos vacío para indicar que el usuario no necesita emplear contraseña en el proceso de identificación. En el tercer campo y el cuarto, con valor '0', son el identificador de usuario y el de grupo, respectivamente. El quinto, con valor 'root', es el denominado 'Gecos' que contiene un comentario descriptivo de la persona o la cuenta. El sexto es el directorio de usuario, en este caso '/root'. El último indica la ruta al programa que se ejecutará cada vez que el usuario se loguee en el sistema, en este caso la Shell '/bin/sh'.

El siguiente paso es crear el fichero '/etc/group' con el siguiente contenido:

```
root:x:0:root
```

Tabla 11: Contenido del fichero 'group'

Los campos que se reflejan son: nombre del grupo ('root' en este caso), contraseña del grupo ('x', con lo que indicamos que el grupo no tiene contraseña), ID del grupo y, por último, lista de usuario que pertenecen al grupo (en este caso únicamente 'root'. Si hubiese más, la lista iría separada por comas).

Una vez creados estos dos ficheros, reiniciamos nuestro sistema embebido y los cambios serán definitivos.

3.8 Arrancando el servidor web incluido en 'busybox'

Este proceso es muy sencillo, pues ya tenemos un servidor Web embebido gracias al rootfs preconfigurado que nos hemos descargado con anterioridad. Para alojar nuestra aplicación Web, crearemos el directorio '/www'. Una vez hecho esto, únicamente debemos lanzar el siguiente comando, que pondrá el servidor Web a escuchar en el puerto 80, y empleará como directorio raíz el que acabamos de crear:

```
$>httpd –p 80 –h /www
```

Tabla 12: Arranque del Servidor Web embebido

3.9 Editando el fichero rcS

Este fichero contiene la secuencia de comandos que se llevan a cabo al inicio del sistema. Podemos emplearlo para que se arranque de forma automática el servidor Web, por ejemplo, y nuestra futura aplicación de monitorización de temperaturas. Ya que esta última aplicación queda descrita en el apartado 4 de este mismo documento, puntualizar que debemos de programarla como demonio del sistema si queremos que corra en segundo plano, es decir, que no bloquee la consola mientras lleva a cabo su cometido, impidiendo la interacción del usuario con el entorno. Para ello, debemos seguir las recomendaciones indicadas en el apartado 3.11.2 de esta memoria.

Como bien comentamos, podemos editar el fichero 'rcS', ubicado en '/etc/init.d/rcS' para agregarle las secuencias que llaman a 'temp_monitoring' y a 'httpd' que podemos ver a continuación, de manera que tanto la aplicación de monitorización de temperaturas (que llamaremos 'temp_monitoring' y se ubicará en la raíz del sistema de archivos), como el servidor Web, se lancen de manera automática al terminar la carga del sistema operativo:

```
#!/bin/sh

/bin/echo "Starting rcS..."

/bin/echo "++ Creating device points"
/bin/mkdir /dev/pts
/bin/mount -t devpts devpts /dev/pts

/bin/echo "++ Mounting filesystem"
/bin/mount -t proc none /proc
/bin/mount -t sysfs none /sys
/bin/mount -t tmpfs none /tmp

/bin/echo "++ Loading system loggers"
/sbin/syslogd
/sbin/klogd

/bin/echo "++ Starting telnet daemon"
/sbin/telnetd -l /bin/sh
```

```
/bin/echo "++ Starting tempterature monitoring daemon"
/temp_monitoring

/bin/echo "++ Starting httpd webserver on port 80 and home dir /www"
/sbin/httpd -p 80 -h /www

/bin/echo "rcS Complete"
```

Tabla 13: Contenido del fichero 'rcS'

3.10 Configurando Eclipse para el desarrollo de aplicaciones

Siempre existen varias formas de proceder ante el desarrollo de aplicaciones, todo depende de las preferencias de cada uno. Se pueden desarrollar directamente sin ningún IDE, creando un fichero Makefile, etc. Sin embargo, el contar con un IDE que nos permita tener los ficheros visualmente organizados, poder contar con un sistema de análisis de código, que genere y actualice por su cuenta los ficheros de construcción de la aplicación, etc., siempre resulta de gran utilidad. En este caso, se ha seleccionado un IDE ampliamente extendido, Eclipse, en su versión Juno para desarrollo de aplicaciones en C/C++. Suponemos que ya se tiene instalado este IDE, y en este apartado únicamente vamos a explicar algunos pequeños pasos y consideraciones para configurarlo con el fin de que compile nuestro Software de forma cruzada, obteniendo un binario para el sistema final, que es Microblaze.

Eclipse nos ofrece una forma muy sencilla para configurar nuestros compiladores cruzados de forma que sean invocados de forma automática. Al crear un nuevo proyecto, por ejemplo en C, seleccionaremos una aplicación vacía, y como toolchain la opción 'Coss GCC':

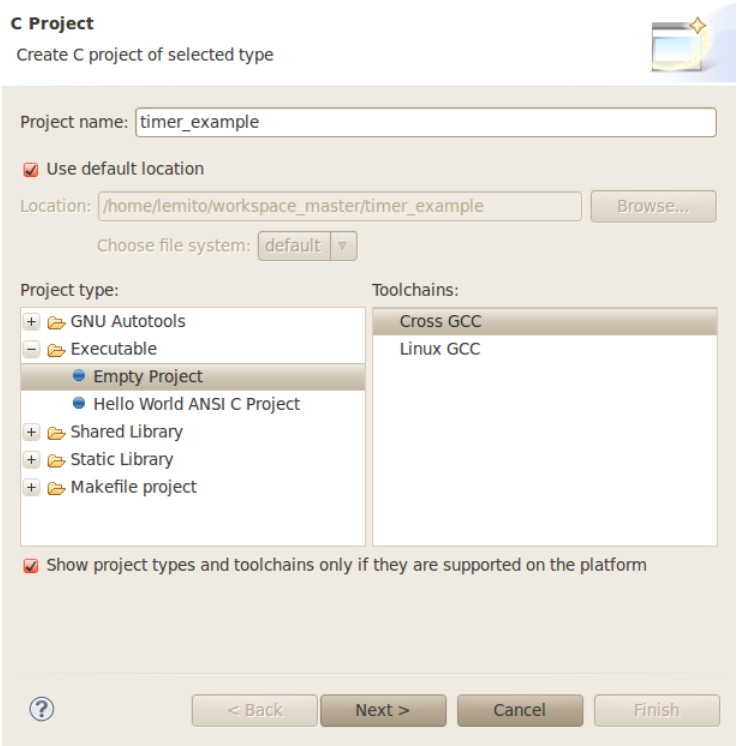


Ilustración 16: Generación de un nuevo proyecto en Eclipse

En la pantalla de configuración, seleccionaremos las que deseemos:

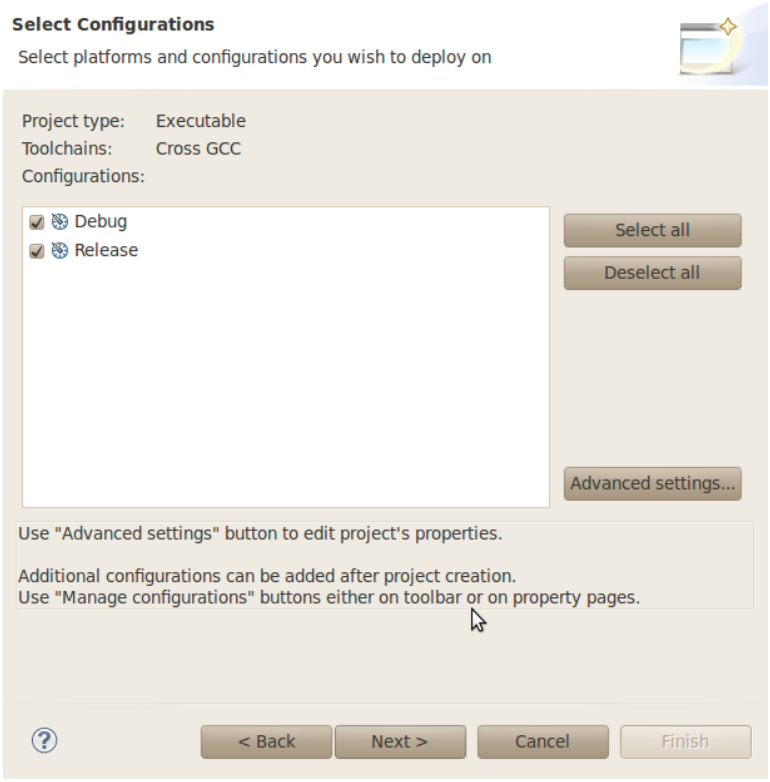


Ilustración 17: Configuraciones del proyecto en Eclipse

Por último, en la última pantalla podemos seleccionar tanto el prefijo para la compilación cruzada, como la ruta a las herramientas del toolchain que nos descargamos desde el repositorio GIT:

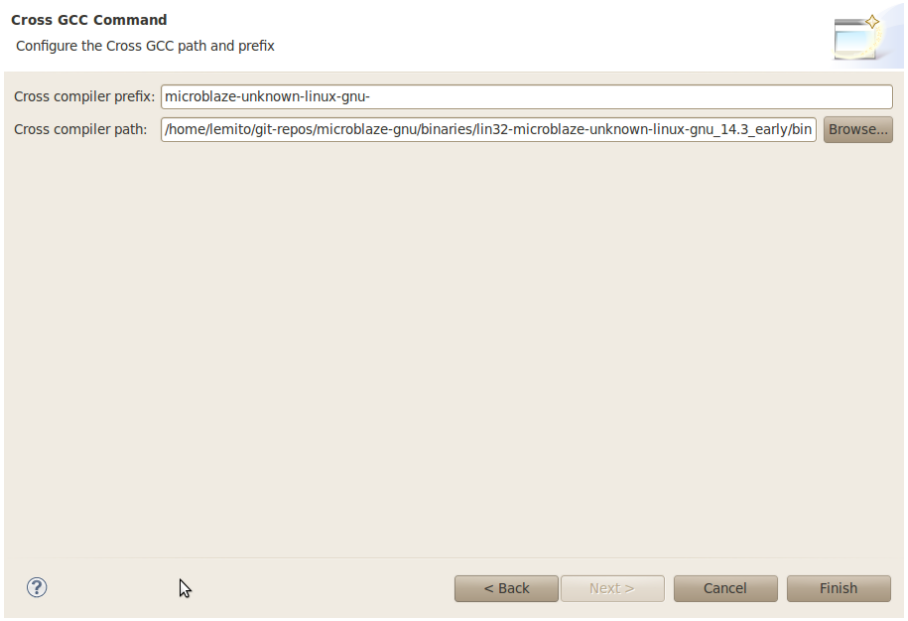


Ilustración 18: Configurando las herramientas para la compilación cruzada en Eclipse

Con estos pasos, ya tenemos las herramientas de compilación cruzada configuradas para Eclipse, de forma que podemos desarrollar y generar los ejecutables para nuestro sistema final como si lo hiciéramos para un PC de escritorio. Únicamente queda destacar otro aspecto. En las propiedades del proyecto, en el apartado 'C/C++ Build → Settings', debemos ir a la opción 'Cross GCC Linker → Miscellaneous', y en el input 'Linker flags' introducir el valor '-static' si queremos que nuestras aplicaciones se compilen de forma estática, es decir, sin hacer uso de librerías dinámicas:

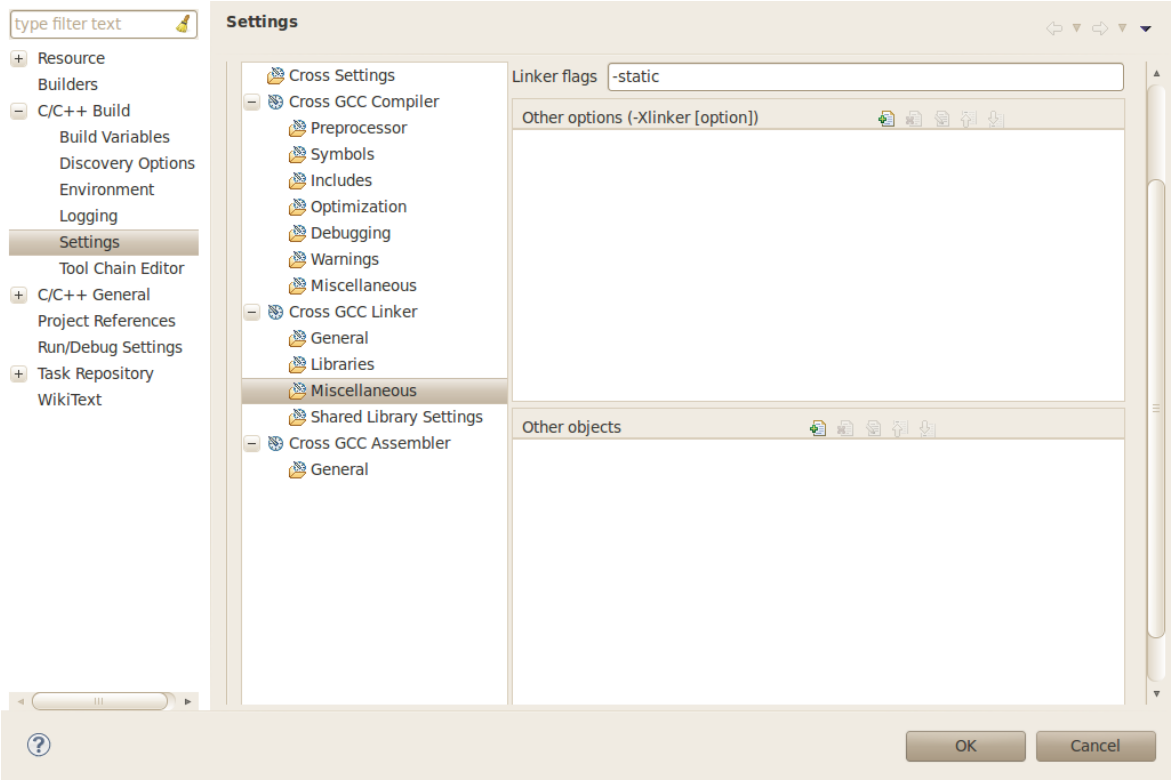


Ilustración 19: Agregando los flags para la compilación estática en Eclipse

3.11 Aplicación de ejemplo

Una vez configurado Eclipse para el desarrollo de nuestras aplicaciones de forma que funcionen sobre Microblaze, vamos a ver una sencilla aplicación de ejemplo. Desarrollaremos el típico 'Hello World!', pero para añadir un poco más de complejidad al asunto, emplearemos timers de forma que se muestre el mensaje 3 veces por pantalla. En nuestro proyecto ya configurado, añadiremos un nuevo fichero fuente llamado 'main.c' que tendrá el siguiente código:

```
#include <time.h>
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

int count;
```

```
void timer_handler(sigval_t val)
{
      printf("Hello embedded Microblaze World!! Counter: %d\r\n", count);
      count++;
}

int main(void)
{
    count = 0;

    struct sigevent evp = { 0 }; // zero all members

    evp.sigev_notify = SIGEV_THREAD;
    evp.sigev_notify_function = timer_handler;
    evp.sigev_value.sival_ptr = (void*)&count;

    struct itimerspec timerspec = { 1, 0, 1, 0 };

    timer_t id;

    timer_create(CLOCK_REALTIME, &evp, &id);
    timer_settime(id, 0, &timerspec, NULL);

    printf("Timer Created\n");

    while (count < 3)
    {
        sleep(1);
    }
    timer_delete(id);
    printf("Timer Destroyed\n");

   return 0;
}
```

Tabla 14: Contenido del fichero 'main.c' de la aplicación de ejemplo

No es objeto de este proyecto exponer el código en detalle. Básicamente, lo que hace el programa es crear un timer que ejecutará la función "timer_handler" tres veces, imprimiendo por pantalla el mensaje "Hello embedded Microblaze World!! Counter: <*valor_contador*>", en el que el valor del contador irá variando. Para más referencias sobre C, y en concreto el estándar POSIX, se recomienda encarecidamente leer la referencia [DR33].

Para configurar Eclipse de forma que compile correctamente nuestras aplicaciones, debemos de linkarle con dos librerías del sistema. Ya que estamos empleando funcionalidades de tiempo y también se deben de crear hilos, hay que habilitar como flags en la etapa de link tanto la librería de hilos 'pthread' como la de funciones relacionadas con el tiempo 'rt'. Para ello, abrimos las opciones de nuestro proyecto, y agregamos los siguientes flags:

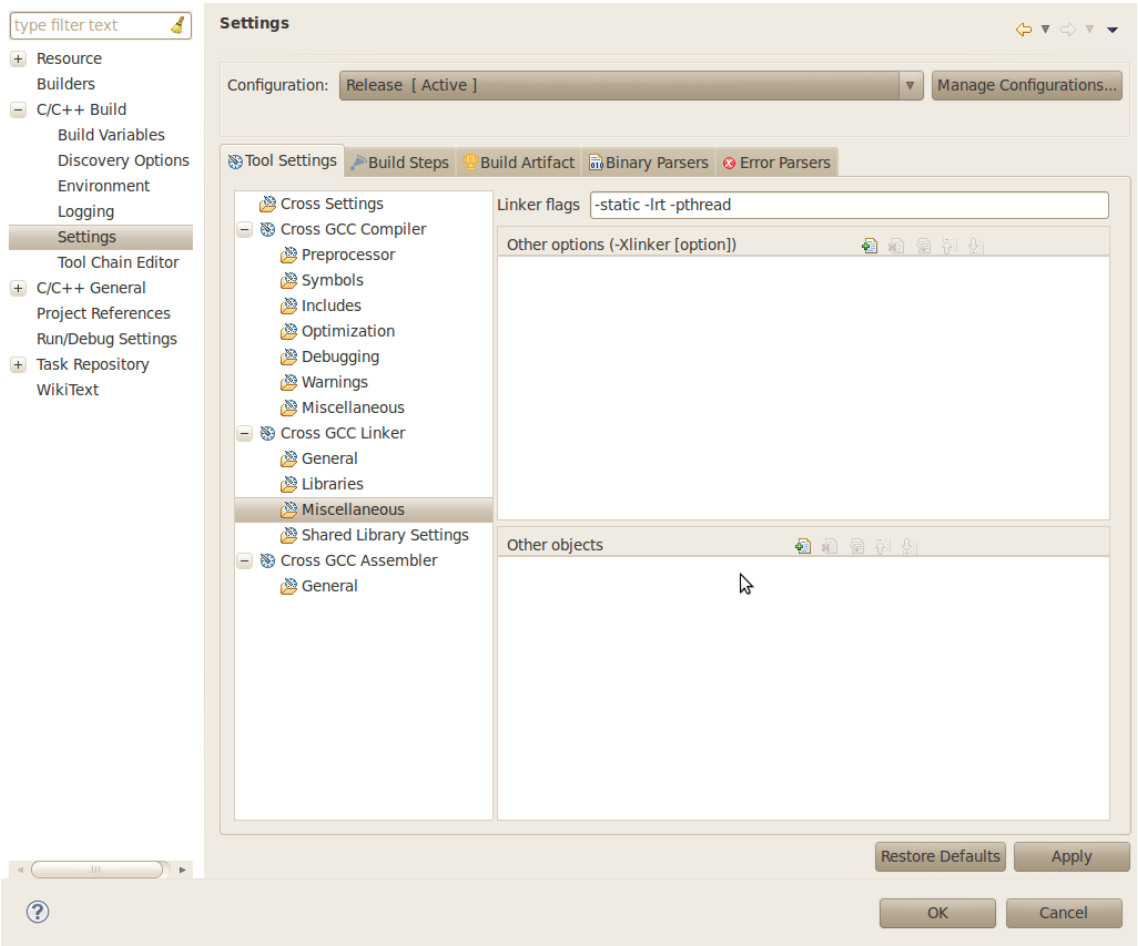


Ilustración 20: Configuración de Eclipse para agregar las librerías para el manejo de hilos y de las funciones de tiempo

Un detalle a tener en cuenta para que nuestra aplicación se compile de forma adecuada [DR34], es la forma en que trabaja el proceso de 'link' o enlazado de la aplicación. El linker va manteniendo, para cada fichero que encuentra en la lista que se le proporciona, una serie de símbolos que necesita. Cuando pasa al siguiente archivo, del elemento anterior únicamente mantiene esta lista para ir buscándolo en los siguientes. Si, por ejemplo, empleásemos este comando:

$> gcc -Wall -o main -lrt main.c

El problema sería el siguiente. El linker miraría que se debe agregar la librería de funciones de tiempo (mediante '-ltr'), y al ver que no tiene ninguna dependencia, continuaría con el procesado. Encontraría 'main.c', analizaría sus dependencias y vería que tiene inclusión de las librerías de funciones de tiempo (time.h). Lo anotaría, pasaría al siguiente y, como no hay ninguno más, esas dependencias no podrían ser resueltas, por lo que el proceso de linkado fallaría. Sin embargo, si lo hacemos de la siguiente forma:

$> gcc -Wall -o main main.c -lrt

Primero analizaría ‘main.c’, anotaría sus símbolos que no ha podido resolver, y al incluir las funcionalidades de tiempo mediante ‘-lrt’, podría resolverlos y enlazar el ejecutable correctamente. Por ello, en Eclipse debemos de indicarle en el patrón de la línea de comandos, el elemento ‘${FLAGS}’ que son los que hemos introducido anteriormente, vaya al final del comando de compilación para que el ejecutable se genere de forma correcta:

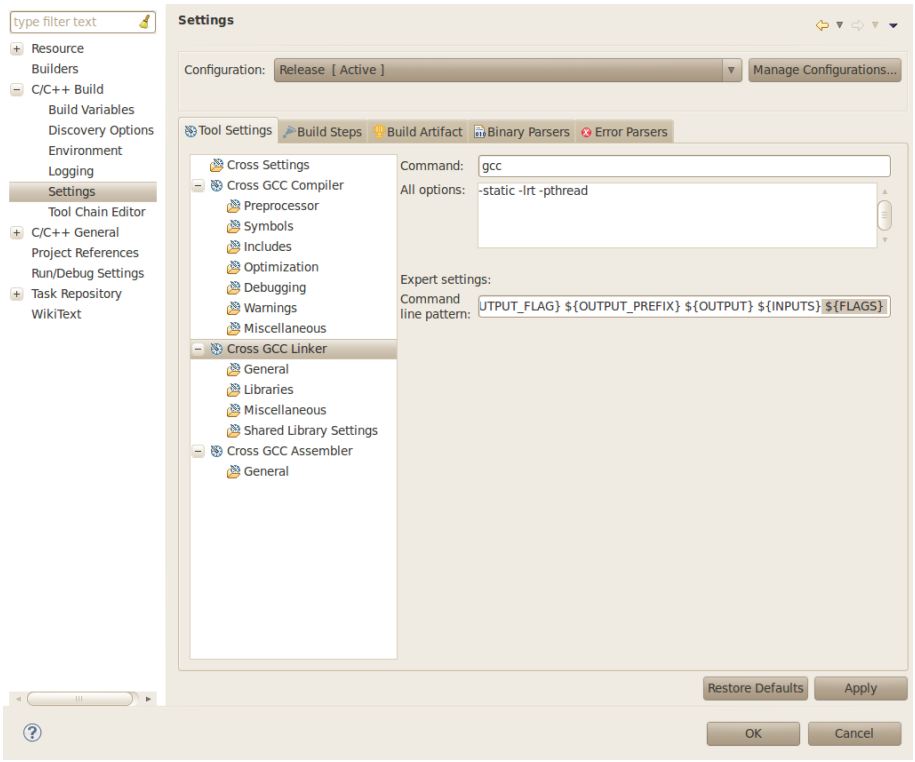


Ilustración 21: Reordenación del patrón de la línea de comandos para el enlazado en Eclipse

Una vez hecho esto, ya podemos compilar correctamente nuestra aplicación. Para verificar que se ha aplicado el proceso de compilación cruzada sobre nuestro fichero ejecutable generado, podemos emplear el comando ‘readelf’ para leer la información del mismo:

```
lemito@lemito-desktop-ubuntu:~/workspace_master/timer_example/Release$ readelf -hd timer_example
ELF Header:
  Magic:   7f 45 4c 46 01 02 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, big endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Xilinx MicroBlaze
  Version:                           0x1
  Entry point address:               0x10002b6c
  Start of program headers:          52 (bytes into file)
  Start of section headers:          823164 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         4
  Size of section headers:           40 (bytes)
  Number of section headers:         37
  Section header string table index: 34

There is no dynamic section in this file.
```

Ilustración 22: Información del fichero binario generado para Microblaze

Como podemos observar, el ejecutable se ha generado para una máquina 'Xilinx Microblaze' y la última línea nos indica que no hay secciones dinámicas en el fichero, lo cual se debe a haberlo compilado todo de forma estática mediante el parámetro '-static'. Si ahora copiamos nuestro fichero ejecutable a la raíz de la carpeta NFS que estamos exportando y montando en el sistema embebido como rootfs, y lo ejecutamos, podemos ver como el resultado obtenido es el esperado:

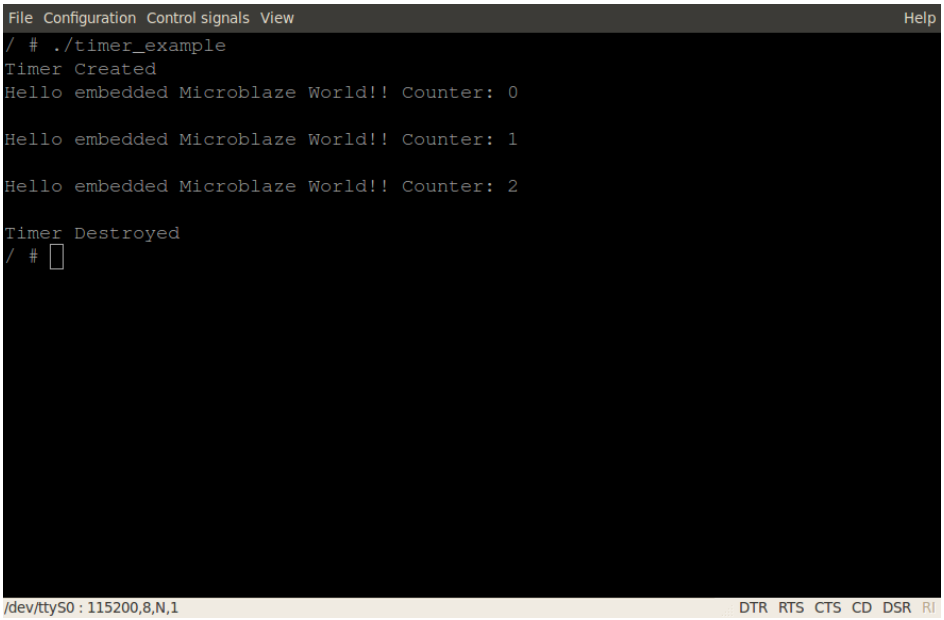


Ilustración 23: Salida obtenida tras ejecutar la aplicación de ejemplo

3.11.1 Programación de aplicaciones como demonios del sistema

Un punto importante en los sistemas embebidos en los que vamos a ejecutar más de una tarea en paralelo es el desarrollo de aplicaciones como demonios. Las aplicaciones que funcionarán como demonios, o *daemons*, son básicamente programas que se ejecutan para llevar a cabo una tarea específica, normalmente repetitiva, y que no van a informar al usuario de manera directa (se suele hacer a través de logs u otros ficheros), y que tampoco van a requerir de su intervención. Para implementar un demonio en Linux, simplemente debemos seguir una sencilla estructura como la que vemos a continuación [DR33]:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
#include <syslog.h>
#include <string.h>

int main(void) {
        /* Our process ID and Session ID */
```

```
        pid_t pid, sid;

        /* Fork off the parent process */
        pid = fork();
        if (pid < 0) {
                exit(EXIT_FAILURE);
        }
        /* If we got a good PID, then
           we can exit the parent process. */
        if (pid > 0) {
                exit(EXIT_SUCCESS);
        }

        /* Change the file mode mask */
        umask(0);

        /* Open any logs here */

        /* Create a new SID for the child process */
        sid = setsid();
        if (sid < 0) {
                /* Log the failure */
                exit(EXIT_FAILURE);
        }

        /* Change the current working directory */
        if ((chdir("/")) < 0) {
                /* Log the failure */
                exit(EXIT_FAILURE);
        }

        /* Close out the standard file descriptors */
        close(STDIN_FILENO);
        close(STDOUT_FILENO);
        close(STDERR_FILENO);

        /* Daemon-specific initialization goes here */

        /* The Big Loop */
        while (1) {
           /* Do some task here ... */

           sleep(30); /* wait 30 seconds */
        }
   exit(EXIT_SUCCESS);
}
```

Tabla 15: Estructura de un demonio Linux en C

En el ejemplo anterior, podemos ver una serie de pasos que se van realizando. Básicamente, consiste en hacer un 'fork' del proceso y, si el código es el del padre, terminar la ejecución. De esta forma, nos quedaremos únicamente con el proceso hijo en memoria. Tras ello, llamaremos a 'umask' para cambiar los permisos que aplique nuestra aplicación a partir de ahora cuando cree entradas en el sistema de ficheros, de forma que se puedan leer y escribir correctamente. Tras ese punto, deberíamos abrir los logs que maneje nuestra aplicación y, tras ello, es el momento de solicitar un ID de sesión único al Kernel (llamando a

‘sid’), consiguiendo así que nuestro proceso no se quede ‘huérfano’ (recordemos que, al hacer el fork, la ejecución del padre finalizó). El siguiente paso sería cambiar el directorio de trabajo a aquél que necesitemos para funcionar, y cerrar los descriptores estándar llamando a ‘close’ (recordemos que un demonio, en teoría, no es cribe en consola, no lee de ella, y no emplea la salida de error, si no que todo se vuelva en un log). Inicializamos los elementos necesarios de nuestra aplicación, y entramos en el bucle infinito en el que llevaremos a cabo nuestra tarea. Con esta sencilla estructura, nuestro programa podrá operar en segundo plano en el sistema sin interferir con otras tareas de usuario.

4 Diseño de la aplicación para la monitorización y visualización de temperaturas

Este apartado expone el diseño de la aplicación que se ha desarrollado para la monitorización remota de temperaturas, así como para su visualización. Todo lo descrito en este epígrafe es un diseño Software que correo sobre un sistema operativo GNU/Linux sobre el Soft-Core Microblaze de Xilinx.

4.1 Descripción del entorno tecnológico

En este apartado describiremos el entorno tecnológico sobre el que se sustenta el prototipo desarrollado. Este entorno lo aproximaremos en forma de capas, tal y como podemos ver en la siguiente imagen:

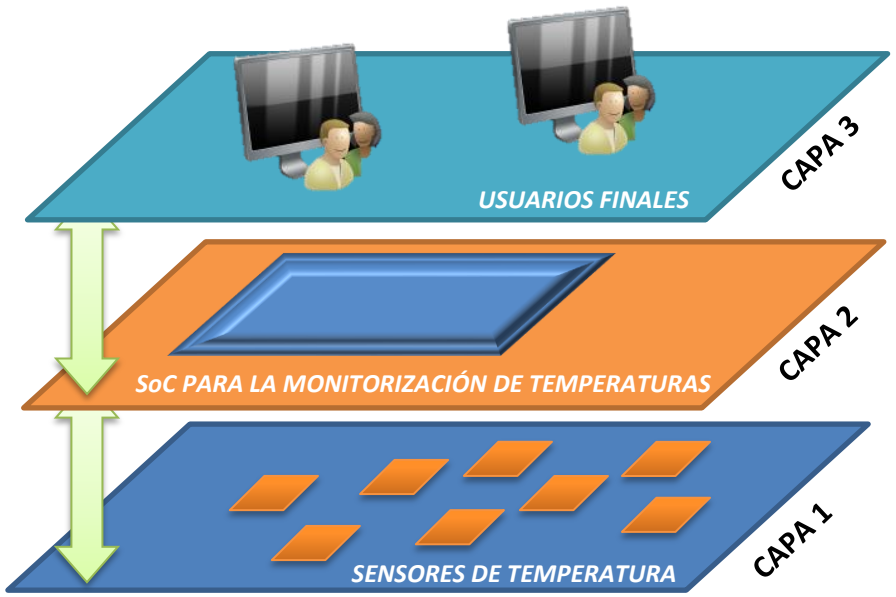


Ilustración 24: Capas que conforman el entorno tecnológico del proyecto

4.1.1 Primera capa

En esta primera capa se encuentran los propios sensores de temperatura. Para el desarrollo del presente prototipo, se ha montado para las pruebas en una protoboard un sistema con 2 sensores LM83 y 3 diodos de temperatura conectados a cada uno, tal y como podemos ver en la siguiente imagen:

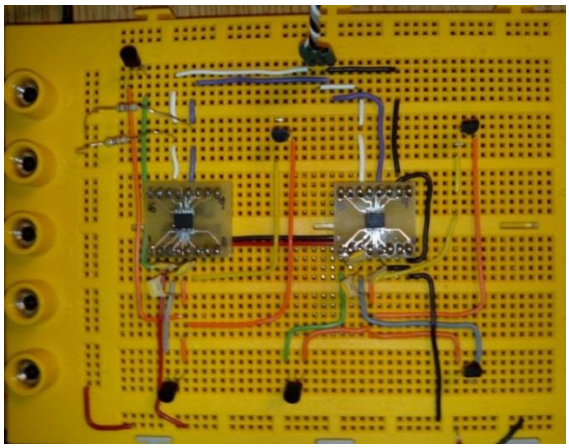


Ilustración 25: Circuito prototipado para montar los sensores de temperatura

Sin embargo, nada impide que estos sensores se encuentren en otro entorno. Por ejemplo, podríamos tenerlos repartidos por un rack de comunicaciones, en varias PCBs, y conectarlos a un bus I2C. Para profundizar más en este aspecto, estas PCBs podrían tener detrás un backplane VME, y dentro de los conectores de este bus se ofrecen una serie de pines libres que el usuario puede emplear, por lo que podrían utilizarse dos de ellos para implementar este bus I2C sobre el backplane, y tener así distribuidos los sensores de temperatura por distintos subsistemas, todos interconectados de forma común. A su vez, este rack podría encontrarse en un armario de comunicaciones con lo que podríamos estar monitorizando las temperaturas de distintos equipos de comunicaciones. O ir montado en una unidad de un UAV y monitorizar información de los distintos componentes del mismo.

4.1.2 Segunda capa

En esta capa se ubica el Sistema Embebido para la monitorización de temperaturas. Como ya se ha descrito, este sistema consta en el prototipo desarrollado de una placa de evaluación Spartan-3AN de Xilinx, la cual podemos ver en la siguiente imagen:

Ilustración 26: Placa de evaluación "Spartan-3AN Starter Kit" empleada en el proyecto

Este es un desarrollo llevado a cabo en forma de prototipo. Sin embargo, si seguimos con los ejemplos expuestos para la primera capa, se podría desarrollar una PCB que contuviese los elementos necesarios (FPGA, memoria RAM, memoria FLASH, conectividad Ethernet, conectividad al bus I2C, etc.) que estuviese conectada al backplane VME para poder acceder a los sensores de temperatura.

En este nivel es donde se implementa el procesador Soft-Core de Microblaze, sobre el que se corre una distribución de Linux compilada para esta arquitectura, y las aplicaciones desarrolladas para monitorizar las temperaturas y ofrecer esta información a los usuarios finales. Este nivel podría, a su vez, subdividirse entonces en otra serie de capas que se comentan a continuación, aunque podemos ver un pequeño esquema en la siguiente imagen:

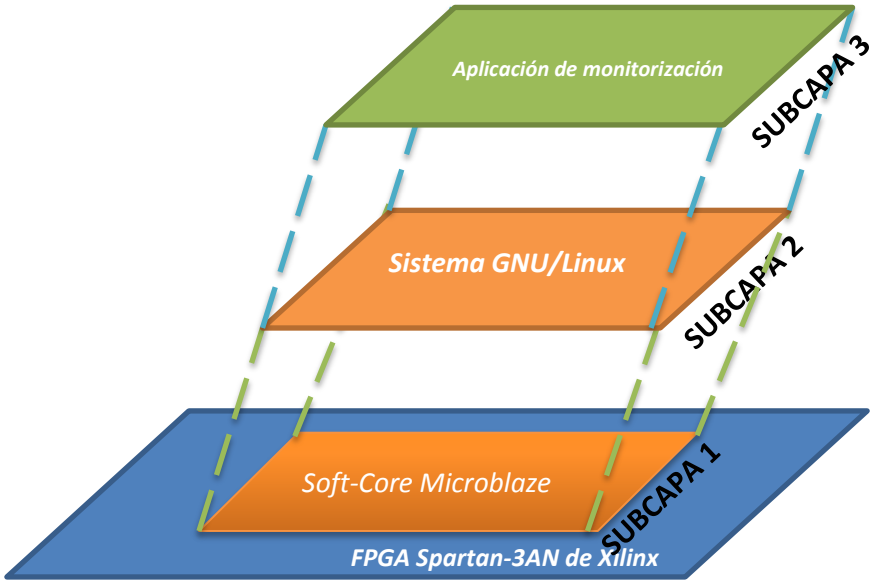


Ilustración 27: Subcapas que conforman el segundo nivel del entorno tecnológico

4.1.2.1 Primera subcapa: FPGA

Este primer nivel es el propiamente físico. En él, tenemos nuestra FPGA de bajo coste, en este caso la familia Spartan-3A de Xilinx. Sobre ella, implementaremos nuestro sistema embebido, tal y como podemos ver en la siguiente imagen:

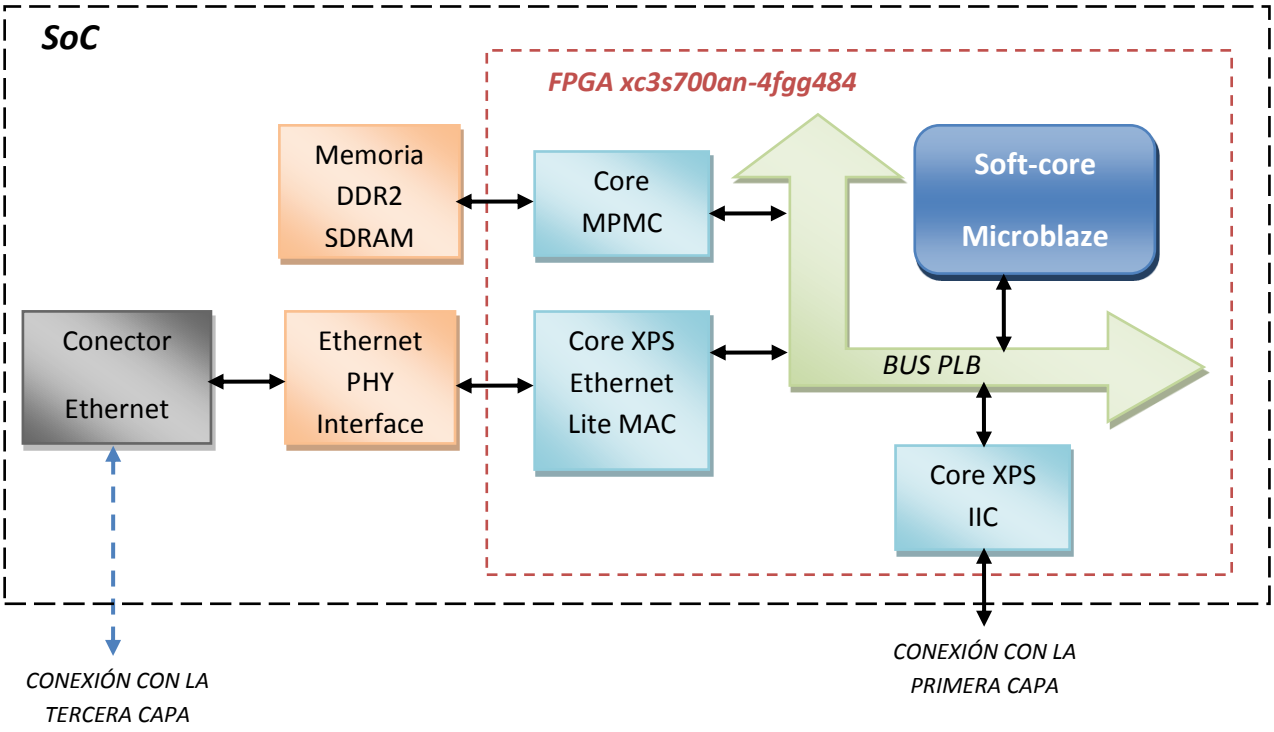


Ilustración 28: Esquema del SoC diseñado

En el entorno reconfigurable se sintetizará un procesador soft-core Microblaze de Xilinx. Además, se empleará la lógica disponible para implementar varios periféricos, entre los que podemos destacar un controlador de memoria multipuerto (MPMC), un periférico para obtener el nivel MAC de la arquitectura de comunicaciones TCP/IP (Ethernet Lite MAC) y un controlador para poder acceder a un bus de comunicaciones IIC (XPS IIC).

4.1.2.2 Segunda subcapa: Sistema operativo GNU/Linux

Sobre nuestro Soft-Core Microblaze de Xilinx haremos correr un sistema operativo de la familia GNU/Linux. Para ello, se han descargado las fuentes del Kernel, configurándose y compilándose para esta arquitectura. Se han activado en el mismo los drivers de acceso al bus I2C de Xilinx, así como las capacidades de conectividad de red para poder ofrecer a través de TCP/IP los datos de las temperaturas en forma de aplicación Web.

4.1.2.3 Tercera subcapa: aplicación de monitorización

En esta tercera subcapa es donde se despliega el Software desarrollado para manejar el prototipo descrito en este documento. Por un lado tenemos un módulo que accede a las

temperaturas ofrecidas por los sensores a través del bus I2C, y por otro una aplicación Web que ofrece de manera remota esa información a los usuarios.

4.1.3 Tercera capa

En esta última capa es donde se ubican los usuarios finales. A través de un PC (o cualquier otra plataforma) que cuente con un navegador para soporte con HTML5 accederán al servidor Web embebido en la segunda capa, el cual les ofrecerá la visualización de los datos de temperaturas que se han muestreado de los sensores. También podrán visualizar el histórico con la información para cada sensor, y modificar la configuración XML de la aplicación.

4.2 Desarrollo modular

Como se ha expuesto, se ha seguido un desarrollo modular que permite adaptar fácilmente el prototipo desarrollado a otros entornos de operación.

Si seguimos explotando los ejemplos puestos en las capas descritas en el entorno tecnológico, el tercer nivel podría, por ejemplo, ser eliminado. A su vez, en el segundo nivel, se podría cambiar la conectividad Ethernet por un core desarrollado a medida que manejase un radioenlace (o que se comunicase con otros subsistemas para pasarle la información de las temperaturas leídas), con lo que podría fácilmente integrarse el prototipo implementado por ejemplo en un UAV y enviar los datos muestreados a una estación en tierra, o almacenarlos para su posterior postprocesado.

Por otro lado, emplear un bus de comunicaciones tan extendido como es I2C nos permitiría, realizando unas mínimas modificaciones en las aplicaciones SW desarrolladas, obtener información de otros sensores, no únicamente sobre temperaturas. Existe un amplio abanico de elementos que se pueden conectar a este bus, como acelerómetros, giroscopios, sensores de humedad, de luz, etc. Ante este planteamiento, el abanico de posibles aplicaciones en las que se podría integrar el presente desarrollo crece exponencialmente.

4.3 Descripción general del módulo de monitorización y visualización de temperaturas

Como ya hemos indicado, la aplicación que monitoriza temperaturas implementada en este prototipo tiene dos partes fundamentales. Por un lado, debemos de obtener la información de los sensores conectados al bus y, por otro, ofrecer esos datos al usuario final para que pueda visualizarlos. El esquema de funcionamiento en el que se unen estas dos partes lo podemos ver a grandes rasgos en la siguiente figura:

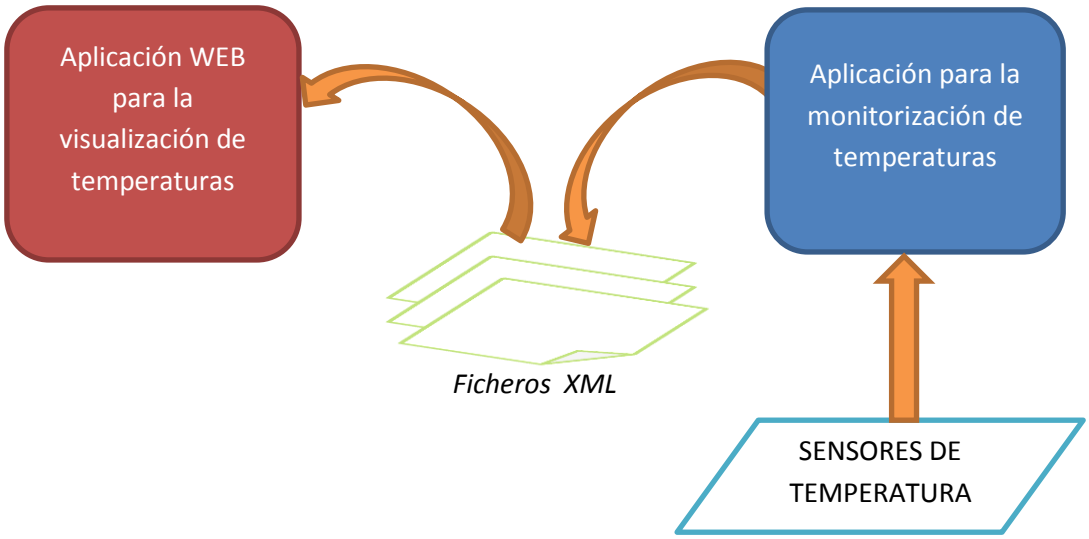


Ilustración 29: Interfaz entre la aplicación de monitorización de temperaturas y la de visualización

Como se puede deducir del esquema anterior, el módulo que se encarga de monitorizar las temperaturas accederá al bus I2C y se comunicará con los sensores para leer esta información. Una vez obtenida, volcará estos datos en una serie de ficheros XML en el sistema de archivos. Por otra parte, la aplicación Web que permite visualizar las temperaturas, accederá a estos ficheros XML para leerlos, y generará la interfaz pertinente con la información contenida en ellos para mostrar al usuario final estos datos.

Con esta aproximación, volvemos a tener un desarrollo modular también en el Software. Se puede sustituir la parte de visualización de las temperaturas por otra aplicación que, por ejemplo, haga uso de un periférico que manda a través de un enlace dedicado estos datos a otro módulo de un sistema, el cual a su vez se encarga de guardar esta información, por ejemplo, en una tarjeta de memoria para ser analizados con posterioridad.

4.4 Requisitos de alto nivel

Este apartado recoge una serie de requisitos de alto nivel que definen, a grandes rasgos, la funcionalidad con la que debe cumplir el prototipo desarrollado y expuesto en el presente documento:

Código	REQ.01
Autor	Eloy Soto Solana
Fecha	12/02/2013
Descripción	El sistema desarrollado debe tener comunicación con sensores de temperatura LM83 conectados a un bus I2C y a sus

	correspondientes diodos.
Comentarios	

Tabla 16: Requisito de alto nivel REQ.01

Código	REQ.02
Autor	Eloy Soto Solana
Fecha	12/02/2013
Descripción	El diseño del prototipo debe ser lo más modular posible, definiendo interfaces de comunicación entre los módulos que lo conformen.
Comentarios	

Tabla 17: Requisito de alto nivel REQ.02

Código	REQ.03
Autor	Eloy Soto Solana
Fecha	12/02/2013
Descripción	Las temperaturas remotas que se lean de los sensores deben de almacenarse, hasta un máximo determinado, mientras dure la ejecución de la aplicación.
Comentarios	

Tabla 18: Requisito de alto nivel REQ.03

Código	REQ.04
Autor	Eloy Soto Solana
Fecha	12/02/2013
Descripción	El sistema debe acceder al bus I2C como maestro para comunicarse con los sensores y obtener información de sus temperaturas.
Comentarios	

Tabla 19: Requisito de alto nivel REQ.04

Código	REQ.05
Autor	Eloy Soto Solana
Fecha	12/02/2013
Descripción	Las últimas temperaturas leídas de cada sensor y diodo deben ser accesibles por el usuario final, quien debe poder visualizarlas.
Comentarios	

Tabla 20: : Requisito de alto nivel REQ.05

Código	REQ.06
Autor	Eloy Soto Solana
Fecha	12/02/2013
Descripción	Además, para cada sensor/diodo, se debe de poder acceder al histórico que muestre todos los datos almacenados por la aplicación para ese sensor/diodo.
Comentarios	

Tabla 21: Requisito de alto nivel REQ.06

Código	REQ.07
Autor	Eloy Soto Solana
Fecha	12/02/2013
Descripción	Cuando el usuario visualice las últimas temperaturas de los sensores, estas deben actualizarse de forma dinámica cada intervalo de muestreo indicado.
Comentarios	

Tabla 22: Requisito de alto nivel REQ.07

Código	REQ.08
Autor	Eloy Soto Solana
Fecha	12/02/2013
Descripción	El usuario debe de poder modificar y configurar ciertos parámetros de la aplicación, como son: intervalo de muestreo, número de muestras a guardar, temperatura alta, temperatura crítica e información de los sensores con sus diodos presentes en el bus.
Comentarios	

Tabla 23: Requisito de alto nivel REQ.08

Código	REQ.09
Autor	Eloy Soto Solana
Fecha	12/02/2013
Descripción	Para cada sensor, se debe de tener en cuenta que puede tener conectado un diodo, dos, los tres o incluso ninguno de ellos.
Comentarios	

Tabla 24: Requisito de alto nivel REQ.09

Código	REQ.10
Autor	Eloy Soto Solana
Fecha	12/02/2013
Descripción	La aplicación debe de indicar al usuario de forma gráfica cuando un sensor ha superado la temperatura alta o crítica establecida.
Comentarios	

Tabla 25: Requisito de alto nivel REQ.10

Código	REQ.11
Autor	Eloy Soto Solana
Fecha	12/02/2013
Descripción	El módulo que lee las temperaturas y el que muestra los datos al usuario deben ser completamente independientes, definiendo un interfaz común de comunicación entre ambos.
Comentarios	

Tabla 26: Requisito de alto nivel REQ.11

Código	REQ.12
Autor	Eloy Soto Solana
Fecha	12/02/2013
Descripción	La configuración de la aplicación debe ser modificable sin tener que recompilar el código. Es decir, se deben de poder añadir nuevos sensores presentes en el bus, o eliminar aquellos que no se encuentren ya en el mismo, sin tener que recompilar ni modificar el código fuente de la misma.
Comentarios	

Tabla 27: Requisito de alto nivel REQ.12

Código	REQ.13
Autor	Eloy Soto Solana
Fecha	12/02/2013
Descripción	La aplicación debe de volcar en un fichero de log información sobre el estado de la misma, o errores que se produzcan de forma que este pueda ser analizado a posteriori para detectar posibles fallos en el funcionamiento.
Comentarios	

Tabla 28: Requisito de alto nivel REQ.13

Código	REQ.14
Autor	Eloy Soto Solana
Fecha	12/02/2013
Descripción	Los datos de temperaturas deben de poder visualizarse de forma independiente de la plataforma en que se lleve a cabo esta inspección.
Comentarios	Para ello, se empleará tecnología Web, de manera que los requisitos de acceso sean lo menos restrictivos posible.

Tabla 29: Requisito de alto nivel REQ.14

Código	REQ.15
Autor	Eloy Soto Solana
Fecha	12/02/2013
Descripción	La monitorización de temperaturas debe ser un proceso que corra en el sistema operativo en segundo plano, y que ofrezca un mínimo o nula información al usuario de forma directa, y sobre todo que no requiera de su intervención para no entorpecer otras tareas del sistema.
Comentarios	

Tabla 30: Requisito de alto nivel REQ.15

Código	REQ.16
Autor	Eloy Soto Solana
Fecha	12/02/2013
Descripción	En la medida de lo posible, la carga de trabajo que conlleva la visualización de los datos de temperaturas debe de distribuirse, delegando todo el proceso que sea posible en los ordenadores de los usuarios finales que acceden a la información, delegando de esta forma carga de trabajo y aligerando así las tareas que debe llevar a cabo el proceso en el sistema prototipo.
Comentarios	Para ello, ya que se va a emplear tecnología Web, se recomienda emplear lenguajes de scripting que se ejecuten en el navegador del cliente, de forma que corra sobre ellos parte del trabajo necesario para visualizar los datos.

Tabla 31: Requisito de alto nivel REQ.16

Código	REQ.17
Autor	Eloy Soto Solana
Fecha	12/02/2013
Descripción	Al inicio del servicio de monitorización de temperaturas, la información almacenada en anteriores ejecuciones debe ser borrada para comenzar desde cero todo el proceso de muestreo y visualización.
Comentarios	

Tabla 32: Requisito de alto nivel REQ.17

4.5 Diseño del modelo de datos

En este apartado se especifica el diseño del modelo de datos empleado por la aplicación. Básicamente, nos encargaremos de definir la información que debe almacenarse en los archivos XML que forman parte de la arquitectura Software implementada. Como ya se ha comentado, estos ficheros sirven de puente para el intercambio de datos entre la aplicación que monitoriza las temperaturas de los sensores conectados al bus I2C, y la que se encarga de mostrar (en forma de aplicación Web) los datos de temperatura muestreados al usuario.

Para dar soporte a todas las necesidades de información que presenta el prototipo, se tienen una serie de almacenes de datos que deben contener lo siguiente: configuración de la aplicación, últimas temperaturas muestreadas e histórico de temperaturas para cada sensor/diodo.

4.5.1 Diseño del almacén de datos para la configuración de la aplicación

Un apartado importante a tener en cuenta es la configuración de la aplicación. Para ello, deben de almacenarse ciertos parámetros que se emplearán durante el funcionamiento del software implementado, y que son:

- **Número de muestras**: Debemos de indicar el número de muestras que se van a almacenar sobre cada sensor/diodo presente en el prototipo. Es decir, este parámetro indica la cantidad de muestras de temperatura que se van a mantener en memoria, descartando las anteriores.
- **Intervalo de muestreo**: Este parámetro indicará cada cuantos segundos se sondearán los elementos del bus para obtener información sobre las temperaturas. Unido al número de muestras, podemos saber la antigüedad que se mostrará en el histórico de temperaturas de cada diodo/sensor (por ejemplo, 100 muestras a un intervalo de 5 segundos, nos dará como resultado que se almacenan las temperaturas obtenidas en los últimos 500 segundos de ejecución). Este parámetro se encuentra restringido a tomar los valores 5, 10, 15 o 20.
- **Temperatura alta**: Este campo contendrá el valor que se considera como temperatura alta. Esta temperatura será aquella a partir de la cual se considera que se está produciendo un calentamiento al que se debería de prestar atención, pero que no resulta crítico.
- **Temperatura crítica**: En este parámetro se almacenará la temperatura que se considera crítica. Es decir, aquella a partir de la cual se considera que el sistema del cual se están leyendo las temperaturas empieza a correr un riesgo de resultar dañado.
- **Información sobre los sensores LM83**: Además de los campos anteriores, debemos indicar en la configuración la información de cada sensor LM83 conectado al bus de comunicaciones I2C. Los datos sobre cada sensor que guardaremos son:
    - **Nombre del sensor**: Nos ayudará a diferenciarlos del resto de sensores LM83.
    - **Descripción**: Para indicar, por ejemplo, la ubicación física del sensor (un sistema, una PCB, etc). Así el usuario podrá visualizar a qué subsistema pertenece la temperatura mostrada.
    - **Dirección**: La dirección del sensor en el bus I2C. Este es el dato que se empleará para direccionar al dispositivo en el bus. Se trata de un byte en notación hexadecimal.
    - **Información sobre los diodos del sensor**: Como ya se ha comentado, cada sensor LM83 puede también proporcionar información de hasta tres temperaturas remotas. Para ello, se le conectan diodos que medirán esas temperaturas, y que el LM83 podrá ofrecer a través del bus I2C. Los datos a almacenar de cada diodo conectado son:

- **Nombre**: Este campo es fijo, y tomará los valores D0, D1 o D2 para identificar a cada sensor.
- **Descripción**: Con este parámetro podremos identificar, por ejemplo, el sistema en el que se encuentra el diodo y del que se están leyendo las temperaturas.
- **Presente**: Cada LM83 puede tener conectados hasta 3 diodos, pero eso no significa que deban estar todos. Este parámetro, que tomará los valores "true/false", nos indicará si el diodo descrito está o no conectado. Es decir, si el sensor LM83 tiene conectado en su entrada este diodo o no.

A continuación mostramos un ejemple de un fichero XML de configuración que se adapta al diseño descrito:

```
<?xml version="1.0"?>
<configuration>
	<samples_quantity>100</samples_quantity>
	<sampling_interval>5</sampling_interval>
	<high_temperature>36</high_temperature>
	<critical_temperature>50</critical_temperature>

	<sensors>
		<sensor name="LM83_1" address="0x2A">
			<description>FPGA1 sensor</description>
			<diode name="D0" present="true">
				<description>LM83_1_PCB1</description>
			</diode>
			<diode name="D1" present="true">
				<description>LM83_1_PCB2</description>
			</diode>
			<diode name="D2" present="true">
				<description>LM83_1_PCB3</description>
			</diode>
		</sensor>

		<sensor name="LM83_2" address="0x18">
			<description>FPGA2 sensor</description>
			<diode name="D0" present="true">
				<description>LM83_2_PCB4</description>
			</diode>
			<diode name="D1" present="true">
				<description>LM83_2_PCB5</description>
			</diode>
			<diode name="D2" present="true">
				<description>LM83_2_PCB6</description>
			</diode>
		</sensor>
	</sensors>

</configuration>
```

Tabla 33: Ejemplo de implementación XML del almacén de datos para la configuración

4.5.2 Diseño del almacén de datos para guardar la información sobre las últimas temperaturas leídas

Este almacén de datos contendrá información sobre la última temperatura leída de cada sensor/diodo presente en el sistema. Para almacenar estos datos, debemos de tener en cuenta los siguientes campos:

- **Elemento al que pertenece la temperatura leída**: Este campo permitirá indicar a qué sensor/diodo pertenece la información de la temperatura que se refleja. Coincide con el campo de descripción en cada sensor/diodo.
- **Temperatura**: Este atributo almacenará el valor leído de temperatura.

Un ejemplo de un fichero XML que se ajusta a este diseño es el siguiente:

```
<?xml version="1.0"?>
<temperatures>
	<temperature>
		<sensor_description>FPGA1 sensor</sensor_description>
		<value>30</value>
	</temperature>
	<temperature>
		<sensor_description>LM83_1_PCB1</sensor_description>
		<value>28</value>
	</temperature>
	<temperature>
		<sensor_description>LM83_1_PCB2</sensor_description>
		<value>28</value>
	</temperature>
	<temperature>
		<sensor_description>LM83_1_PCB3</sensor_description>
		<value>28</value>
	</temperature>
	<temperature>
		<sensor_description>FPGA2 sensor</sensor_description>
		<value>28</value>
	</temperature>
	<temperature>
		<sensor_description>LM83_2_PCB4</sensor_description>
		<value>28</value>
	</temperature>
	<temperature>
		<sensor_description>LM83_2_PCB5</sensor_description>
		<value>29</value>
	</temperature>
	<temperature>
		<sensor_description>LM83_2_PCB6</sensor_description>
		<value>29</value>
	</temperature>
</temperatures>
```

Tabla 34: Ejemplo de implementación XML para el almacén de datos de últimas temperaturas

4.5.3 Diseño del almacén de datos para guardar el histórico de temperaturas sobre cada sensor/diodo

Por último, necesitamos describir el almacén de datos que contendrá el histórico de las temperaturas tomadas para cada sensor/diodo presente en el sistema. Existirá un fichero XML que se ajusta a este diseño para cada sensor/diodo existente. Por lo tanto, tendremos tantos ficheros XML como sensores y diodos haya en el bus, y cada archivo tendrá como nombre el campo de descripción del sensor al que se asocia. La información que se guarda es:

- **Descripción**: Este campo contiene la descripción del sensor/diodo.
- **Temperaturas**: Valores de las temperaturas tomadas. Tendremos como mucho una lista, separada por comas, de tantos elementos como '**número de muestras**' se indique en el fichero XML de configuración.

A continuación podemos ver el ejemplo de un fichero XML que se ajusta a este diseño:

```
<?xml version="1.0"?>
<sensor>
	<sensor_description>LM83_1_PCB1</sensor_description>
	<temperatures>27,28,28,28,28,28,28,28,28,28,28,28,28,28,28,28,28,28,28
,28,28,28,28,27,28,28,28,28,28,28,28,28,28,28,28,28,28,28,28,28,28,28,28,28,28,2
8,28,28,28,27,28,28,28,28,28,28,28,28,28,28,28,28,28,27,28,28,28,28,28,28,27,
28,28,28,28,28,28,28,28,28,28,27,28,28,28,28,28,28,28,28,28,28,28,28,28,28,28,28
,28,28,28,28</temperatures>
</sensor>
```

Tabla 35: Ejemplo de implementación XML para el almacén de datos del histórico de temperaturas de un sensor

4.6 Diseño de clases para la aplicación de monitorización de temperaturas

En este apartado se analiza el diseño de clases UML que conforma la aplicación de monitorización de temperaturas.

4.6.1 Diagrama general de clases

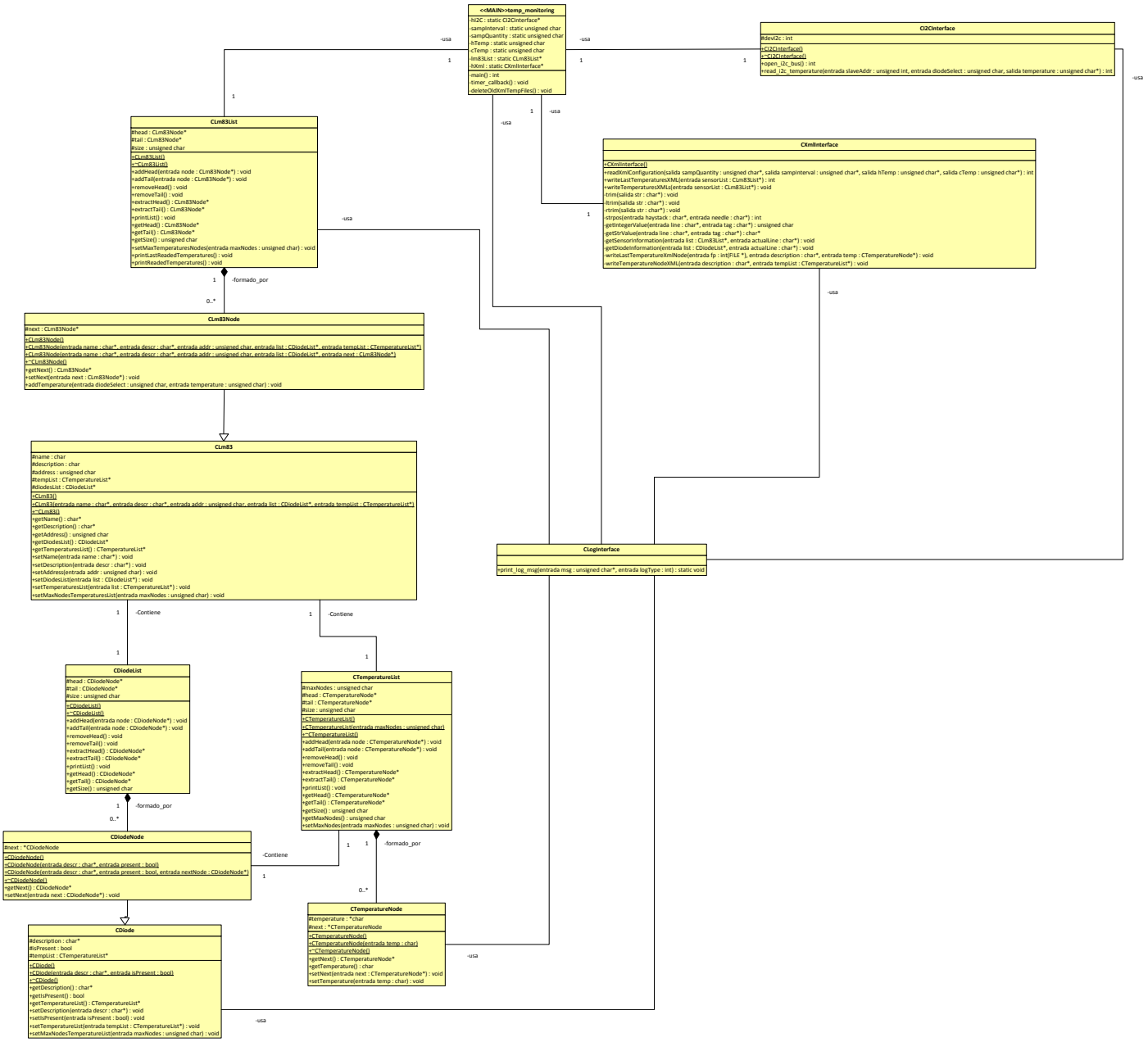


Ilustración 30: Diagrama general de clases

En el esquema anterior podemos ver las distintas relaciones que existen entre las clases que conforman la aplicación de monitorización de temperaturas.

Como vemos, la clase principal 'temp_monitoring' hace uso de las clases 'CI2CInterface' para el control del bus I2C, 'CXmlInterface' para operar sobre los ficheros XML,

‘CLm83List’ para manejar la lista con la información de los sensores/diodos de los que se leen las temperaturas, y ‘CLogInterface’ para producir mensajes de log.

La clase ‘CI2CInterface’ nos permite manejar el bus de comunicaciones I2C, abriendo una conexión con el mismo, u obteniendo datos de los sensores que están conectados a él. Hace uso de la clase de log para informar de situaciones especiales.

‘CXmlInterface’ aísla al resto de la aplicación del manejo de los ficheros XML, como pueden ser la lectura del fichero de configuración o la generación de los archivos que contienen los datos sobre las temperaturas leídas de los sensores. Emplea a su vez la clase ‘CLogInterface’ para generar mensajes de log.

La clase ‘CLm83List’ representa una lista con los sensores que maneja la aplicación. Está formada por nodos ‘CLm83Node’ que heredan sus propiedades y atributos de la clase ‘CLm83’, la cual describe un sensor de temperatura LM83. Esta última clase contiene, a su vez, una lista de diodos y otra de temperaturas. La primera es un objeto de tipo ‘CDiodeList’, que modela y representa una lista de los diodos que puede tener conectados un sensor LM83. Los nodos de esta lista de tipo ‘CDiodeNode’ heredan sus métodos y atributos de la clase ‘CDiode’ (que, a su vez, emplea la clase ‘CTemperatureList’ para contener una lista con las temperaturas que han sido leídas de cada diodo). Por otro lado, el atributo de cada sensor Lm83 de tipo ‘CTemperatureList’ contiene una lista de las temperaturas locales leídas en ese sensor. Esta lista se compone de elementos ‘CTemperatureNode’ que contiene la información de cada dato sondeado.

Para terminar, tanto la clase ‘CDiode’ como ‘CTemperatureNode’ emplean las falibilidades que ofrece ‘CLogInterface’ para volcar información en el fichero de log del sistema. Esta última clase contiene un único método estático para llevar a cabo esta tarea, que es el que permite insertar el mensaje de log junto a la criticidad de dicho mensaje.

4.6.2 Clase “CTemperatureNode”

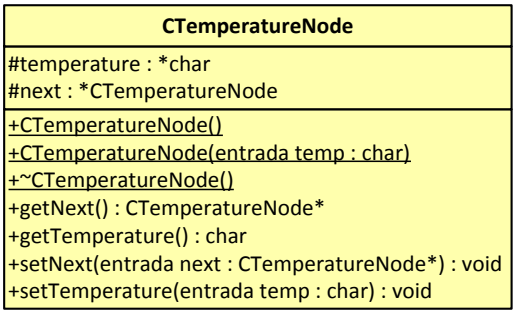


Ilustración 31: Diagrama UML de la clase 'CTemperatureNode'

4.6.2.1 Descripción

Esta clase representa un dato de temperatura obtenido desde un sensor/diodo remoto. Se ha implementado en forma de nodo de una lista, para poder enlazar un conjunto de datos leídos.

4.6.2.2 Atributos

- **Temperature**: Atributo protegido que contiene el dato de la temperatura leída.
- **Next**: Atributo protegido que es un puntero a un elemento de tipo 'CTemperatureNode'. Se encarga de apuntar al siguiente elemento en la lista.

4.6.2.3 Métodos

- **CTemperatureNode:** Constructor de la clase sin parámetros.
- **CTemperatureNode:** Otro constructor de la clase, que recibe como argumento el dato de temperatura y pone el atributo 'temperature' a dicho valor.
- **~CTemperatureNode:** Destructor de la clase.
- **getNext:** Método que nos retorna el atributo 'next' de la clase.
- **getTemperature:** Método que retorna el valor del atributo 'temperature'.
- **setNext:** Recibe como argumento un puntero a un elemento de tipo 'CTemperatureNode', y hace que el atributo 'next' apunte a dicho elemento.
- **setTemperature:** Método que recibe como parámetro un valor de tipo 'char' pone el atributo 'temperature' a dicho valor.

4.6.3 Clase "CTemperatureList"

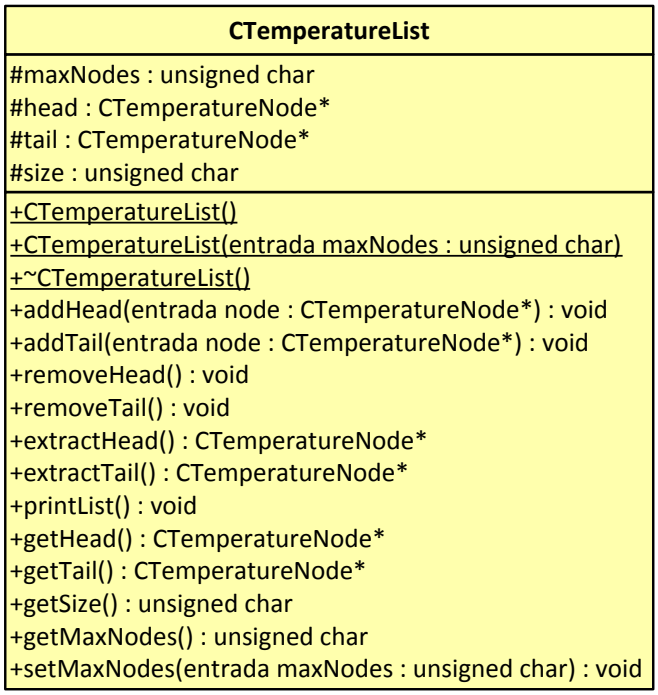


Ilustración 32: Diagrama UML de la clase 'CTemperatureList'

4.6.3.1 Descripción

Esta clase representa un listado de temperaturas leídas de un sensor/diodo.

4.6.3.2 Atributos

- **maxNodes**: Atributo protegido que indica el máximo número de nodos que contendrá la lista. Dado que no tenemos memoria 'ilimitada' y nos encontramos en un sistema embebido, debemos de limitar el número de muestras tomadas de cada sensor/diodo.
- **head**: Atributo protegido que es un puntero al primer nodo de la lista.
- **tail**: Atributo protegido que es un puntero al último elemento de la lista.
- **size**: Atributo protegido que contiene el número de elementos presentes en la lista.

4.6.3.3 Métodos

- **CTemperatureList:** Constructor de la clase sin parámetros.

- **CTemperatureList:** Otro constructor de la clase, que recibe como argumento el máximo número de nodos que contendrá la lista.
- **~CTemperatureNode:** Destructor de la clase.
- **addHead:** Método que inserta un nodo recibido como argumento al inicio de la lista.
- **addTail:** Método que inserta un nodo recibido como argumento al final de la lista.
- **removeHead:** Método que elimina el primer elemento de la lista.
- **removeTail:** Método que elimina el último elemento de la lista.
- **extractHead:** Método que extrae el primer elemento de la lista, y lo retorna. Es decir, quita el primer elemento de la lista, y nos retorna un puntero a ese nodo.
- **extractTail:** Método que extrae el último elemento de la lista, y lo retorna. Es decir, quita el último elemento de la lista, y nos retorna un puntero a ese nodo.
- **printList:** Método que imprime por consola la lista de temperaturas. Está pensado para hacer debug en la aplicación.
- **getHead:** Retorna un puntero al primer elemento de la lista.
- **getTail:** Retorna un puntero al último elemento de la lista.
- **getSize:** Retorna el número de elemento de la lista.
- **getMaxNodes:** Retorna el máximo número de nodos que puede contener la lista.
- **setMaxNodes:** Método que nos permite dar un valor al atributo 'maxNodes' de la clase.

4.6.4 Clase "CDiode"

CDiode
#description : char* #isPresent : bool #tempList : CTemperatureList*
+CDiode() +CDiode(entrada descr : char*, entrada isPresent : bool) +~CDiode() +getDescription() : char* +getIsPresent() : bool +getTemperatureList() : CTemperatureList* +setDescription(entrada descr : char*) : void +setIsPresent(entrada isPresent : bool) : void +setTemperatureList(entrada tempList : CTemperatureList*) : void +setMaxNodesTemperatureList(entrada maxNodes : unsigned char) : void

Ilustración 33: Diagrama UML de la clase 'CDiode'

4.6.4.1 Descripción

Esta clase representa un diodo de temperatura que se encuentra conectado a un sensor LM83.

4.6.4.2 Atributos

- **Description**: Atributo protegido que contiene una descripción del diodo. Por ejemplo, podría contener un texto que nos indique la ubicación del diodo para saber que temperatura se está leyendo.
- **isPresent**: Booleano que nos indica si el diodo se encuentra conectado o no. Un sensor LM83 puede tener conectados hasta 3 diodos, pero no tienen por qué estar todos presentes.
- **tempList**: Atributo que contiene la lista de temperaturas leídas para este diodo en caso de que se encuentre presente.

4.6.4.3 Métodos

- **CDiode:** Constructor de la clase sin parámetros.
- **CDiode:** Otro constructor de la clase que recibe parámetros para inicializar los atributos del objeto.
- **~CDiode:** Destructor de la clase.
- **getDescription:** Método que retorna el valor del atributo 'description'.
- **getIsPresent**: Retorna el valor del atributo 'isPresent'.
- **getTemperatureList:** Retorna un puntero al atributo 'tempList'.
- **setDescription**: Método que nos permite dar un valor al atributo 'description'.
- **setIsPresent:** Este método nos permite dar un valor al atributo 'isPresent'.
- **setTemperatureList**: Método que nos permite establecer el atributo 'tempList'.
- **setMaxNodesTemperaturesList:** Método que se encarga de inicializar el máximo número de nodos que puede contener la lista de temperaturas.

4.6.5 Clase "CDiodeNode"

CDiodeNode
#next : *CDiodeNode
+CDiodeNode() +CDiodeNode(entrada descr : char*, entrada present : bool) +CDiodeNode(entrada descr : char*, entrada present : bool, entrada nextNode : CDiodeNode*) +~CDiodeNode() +getNext() : CDiodeNode* +setNext(entrada next : CDiodeNode*) : void

Ilustración 34: Diagrama UML de la clase 'CDiodeNode'

4.6.5.1 Descripción

Esta clase representa un nodo en una lista de diodos. Esta clase presente herencia de la clase 'CDiode', por lo que hereda todos sus métodos/atributos.

4.6.5.2 Atributos

- **next**: Atributo protegido que contiene un puntero al siguiente nodo de la lista.

4.6.5.3 Métodos

- **CDiodeNode:** Constructor de la clase sin parámetros.
- **CDiodeNode:** Otro constructor de la clase que recibe parámetros para inicializar los atributos del objeto.
- **~CDiodeNode:** Destructor de la clase.
- **getNext:** Método que retorna un puntero al siguiente elemento de la lista.
- **setNext**: Método que nos permite establecer el valor del atributo 'next'.

4.6.6 Clase "CDiodeList"

CDiodeList
#head : CDiodeNode* #tail : CDiodeNode* #size : unsigned char
+CDiodeList() +~CDiodeList() +addHead(entrada node : CDiodeNode*) : void +addTail(entrada node : CDiodeNode*) : void +removeHead() : void +removeTail() : void +extractHead() : CDiodeNode* +extractTail() : CDiodeNode* +printList() : void +getHead() : CDiodeNode* +getTail() : CDiodeNode* +getSize() : unsigned char

Ilustración 35: Diagrama UML de la clase 'CDiodeList'

4.6.6.1 Descripción

Esta clase representa una lista con los diodos conectados a un sensor LM83.

4.6.6.2 Atributos

- **head**: Atributo protegido que es un puntero al primer nodo de la lista.
- **tail**: Atributo protegido que es un puntero al último elemento de la lista.
- **size**: Atributo protegido que contiene el número de elementos presentes en la lista.

4.6.6.3 Métodos

- **CDiodeList:** Constructor de la clase sin parámetros.
- **~CDiodeList:** Destructor de la clase.
- **addHead:** Método que inserta un nodo recibido como argumento al inicio de la lista.
- **addTail:** Método que inserta un nodo recibido como argumento al final de la lista.
- **removeHead:** Método que elimina el primer elemento de la lista.
- **removeTail:** Método que elimina el último elemento de la lista.
- **extractHead:** Método que extrae el primer elemento de la lista, y lo retorna. Es decir, quita el primer elemento de la lista, y nos retorna un puntero a ese nodo.
- **extractTail:** Método que extrae el último elemento de la lista, y lo retorna. Es decir, quita el último elemento de la lista, y nos retorna un puntero a ese nodo.
- **printList:** Método que imprime por consola la lista de diodos con su información. Está pensado para hacer debug en la aplicación.
- **getHead:** Retorna un puntero al primer elemento de la lista.
- **getTail:** Retorna un puntero al último elemento de la lista.
- **getSize:** Retorna el número de elemento de la lista.

4.6.7 Clase "CLm83"

CLm83
#name : char #description : char #address : unsigned char #tempList : CTemperatureList* #diodesList : CDiodeList*
+CLm83() +CLm83(entrada name : char*, entrada descr : char*, entrada addr : unsigned char, entrada list : CDiodeList*, entrada tempList : CTemperatureList*) +~CLm83() +getName() : char* +getDescription() : char* +getAddress() : unsigned char +getDiodesList() : CDiodeList* +getTemperaturesList() : CTemperatureList* +setName(entrada name : char*) : void +setDescription(entrada descr : char*) : void +setAddress(entrada addr : unsigned char) : void +setDiodesList(entrada list : CDiodeList*) : void +setTemperaturesList(entrada list : CTemperatureList*) : void +setMaxNodesTemperaturesList(entrada maxNodes : unsigned char) : void

Ilustración 36: Diagrama UML de la clase 'CLm83'

4.6.7.1 Descripción

Esta clase representa un sensor de temperatura LM83.

4.6.7.2 Atributos

- **Name**: Atributo protegido que contiene el nombre que deseamos darle al sensor para identificarlo
- **Description**: Atributo protegido que contiene una descripción del sensor, por ejemplo para indicar la temperatura que se está midiendo, o el sistema físico (una PCB por ejemplo) donde se encuentra ubicado.
- **address**: Atributo protegido que contiene la dirección del diodo. Esta dirección es la que se empleará para comunicarnos con él a través del bus I2C.
- **tempList**: Atributo que contiene la lista de temperaturas leídas para este sensor (temperaturas locales del sensor, no las remotas de los diodos).
- **diodeList**: Atributo protegido que contiene la lista de diodos en el sensor (3 diodos, estén presentes o no).

4.6.7.3 Métodos

- **CLm83:** Constructor de la clase sin parámetros.
- **CLm83:** Otro constructor de la clase que recibe parámetros para inicializar los atributos del objeto.
- **~CLM83:** Destructor de la clase.
- **getName:** Método de la clase que nos permite obtener el valor del atributo 'name'.
- **getDescription:** Método que retorna el valor del atributo 'description'.
- **getAddress**: Retorna el valor del atributo 'address'.
- **getDiodesList:** Retorna un puntero a la lista de diodos del sensor.
- **getTemperatureList:** Retorna un puntero al atributo 'tempList'.
- **setName:** Este método nos permite establecer al valor del atributo 'name' de la clase.
- **setDescription**: Método que nos permite dar un valor al atributo 'description'.
- **setAddress:** Este método nos permite dar un valor al atributo 'address'.
- **setDiodesList:** Método que nos permite dar un valor al atributo 'diodesList'.
- **setTemperatureList**: Método que nos permite establecer el atributo 'tempList'.
- **setMaxNodesTemperaturesList:** Método que se encarga de inicializar el máximo número de nodos que puede contener la lista de temperaturas del objeto.

4.6.8 Clase "CLm83Node"

CLm83Node
#next : CLm83Node*
+CLm83Node() +CLm83Node(entrada name : char*, entrada descr : char*, entrada addr : unsigned char, entrada list : CDiodeList*, entrada tempList : CTemperatureList*) +CLm83Node(entrada name : char*, entrada descr : char*, entrada addr : unsigned char, entrada list : CDiodeList*, entrada next : CLm83Node*) +~CLm83Node() +getNext() : CLm83Node* +setNext(entrada next : CLm83Node*) : void +addTemperature(entrada diodeSelect : unsigned char, entrada temperature : unsigned char) : void

Ilustración 37: Diagrama UML de la clase 'CLm83Node'

4.6.8.1 Descripción

Esta clase representa un nodo en una lista de sensores LM83. Esta clase presenta herencia de la clase 'CLm83', por lo que hereda todos sus métodos/atributos.

4.6.8.2 Atributos

- **next**: Atributo protegido que contiene un puntero al siguiente nodo de la lista.

4.6.8.3 Métodos

- **CLm83Node:** Constructor de la clase sin parámetros.
- **CLm83Node:** Otro constructor de la clase que recibe parámetros para inicializar los atributos del objeto.
- **CLm83Node:** Otro constructor de la clase que recibe parámetros para inicializar ciertos atributos del objeto.
- **~CLm83Node:** Destructor de la clase.
- **getNext:** Método que retorna un puntero al siguiente elemento de la lista.
- **setNext**: Método que nos permite establecer el valor del atributo 'next'.
- **addTemperature:** Método que recibe como argumentos un entero que identifica si queremos operar sobre el sensor local (LM83) o sobre alguno de los tres diodos que puede tener conectados y un dato de temperatura, y se encarga de agregar esa información a la lista de temperaturas del sensor/diodo indicado.

4.6.9 Clase "CLm83List"

CLm83List
#head : CLm83Node* #tail : CLm83Node* #size : unsigned char
+CLm83List() +~CLm83List() +addHead(entrada node : CLm83Node*) : void +addTail(entrada node : CLm83Node*) : void +removeHead() : void +removeTail() : void +extractHead() : CLm83Node* +extractTail() : CLm83Node* +printList() : void +getHead() : CLm83Node* +getTail() : CLm83Node* +getSize() : unsigned char +setMaxTemperaturesNodes(entrada maxNodes : unsigned char) : void +printLastReadedTemperatures() : void +printReadedTemperatures() : void

Ilustración 38: Diagrama UML de la clase 'CLm83List'

4.6.9.1 Descripción

Esta clase representa una lista con la información de los sensores LM83 a los que accederá la aplicación de monitorización de temperaturas.

4.6.9.2 Atributos

- **head**: Atributo protegido que es un puntero al primer nodo de la lista.
- **tail**: Atributo protegido que es un puntero al último elemento de la lista.
- **size**: Atributo protegido que contiene el número de elementos presentes en la lista.

4.6.9.3 Métodos

- **CLm83List:** Constructor de la clase sin parámetros.
- **~CLm83List:** Destructor de la clase.
- **addHead:** Método que inserta un nodo recibido como argumento al inicio de la lista.
- **addTail:** Método que inserta un nodo recibido como argumento al final de la lista.
- **removeHead:** Método que elimina el primer elemento de la lista.
- **removeTail:** Método que elimina el último elemento de la lista.
- **extractHead:** Método que extrae el primer elemento de la lista, y lo retorna. Es decir, quita el primer elemento de la lista, y nos retorna un puntero a ese nodo.
- **extractTail:** Método que extrae el último elemento de la lista, y lo retorna. Es decir, quita el último elemento de la lista, y nos retorna un puntero a ese nodo.
- **printList:** Método que imprime por consola la lista de sensores con su información. Está pensado para hacer debug en la aplicación.
- **getHead:** Retorna un puntero al primer elemento de la lista.
- **getTail:** Retorna un puntero al último elemento de la lista.
- **getSize:** Retorna el número de elemento de la lista.
- **setMaxTemperaturesNodes:** Método que se encarga de inicializar el máximo número de nodos que puede contener la lista de temperaturas del objeto.
- **printLastReadedtemperatures:** Método que muestra por pantalla las últimas temperaturas leídas para el sensor y sus diodos.
- **printReadedTemperatures:** Método que muestra por pantalla todas las temperaturas leídas para el sensor LM83 y sus diodos.

4.6.10 Clase "CXmlInterface"

CXmlInterface
+CXmlInterface() +readXmlConfiguration(salida sampQuantity : unsigned char*, salida sampInterval : unsigned char*, salida hTemp : unsigned char*, salida cTemp : unsigned char*) : int +writeLastTemperaturesXML(entrada sensorList : CLm83List*) : int +writeTemperaturesXMLs(entrada sensorList : CLm83List*) : void -trim(salida str : char*) : void -ltrim(salida str : char*) : void -rtrim(salida str : char*) : void -strpos(entrada haystack : char*, entrada needle : char*) : int -getIntegerValue(entrada line : char*, entrada tag : char*) : unsigned char -getStrValue(entrada line : char*, entrada tag : char*) : char* -getSensorInformation(entrada list : CLm83List*, entrada actualLine : char*) : void -getDiodeInformation(entrada list : CDiodeList*, entrada actualLine : char*) : void -writeLastTemperatureXmlNode(entrada fp : int(FILE *), entrada description : char*, entrada temp : CTemperatureNode*) : void -writeTemperatureNodeXML(entrada description : char*, entrada tempList : CTemperatureList*) : void

Ilustración 39: Diagrama UML de la clase 'CXmlInterface'

4.6.10.1 Descripción

Esta clase nos ofrece un interfaz para acceder a las funciones de manejo de ficheros XML, abstrayendo al resto de la aplicación de esta tarea.

4.6.10.2 Métodos

- **CXmlInterface:** Constructor de la clase sin parámetros.
- **readXmlConfiguration:** Método público que recibe una serie de argumentos sobre los que volcará la información parseada desde el fichero XML de configuración. Estos datos son la información sobre los sensores LM83 y sus diodos, así como los datos de temperatura máxima, temperatura crítica, intervalo de muestreo y número de muestra a tomar de cada sensor/diodo.
- **writeLastTemperaturesXML:** Método público que se encarga de generar el fichero XML que contiene los datos de las últimas temperaturas leídas desde los sensores. Recibe como argumento para ello la lista de sensores LM83 que maneja la aplicación.
- **writeTemperaturesXMLs:** Método público que se encarga de generar los ficheros XML (uno por cada sensor y diodo existente) que contiene la información de todas las temperaturas leídas para el sensor/diodo al que pertenece cada fichero generado.
- **trim:** Método privado que elimina espacios en blanco y tabuladores del final y del inicio de una cadena pasada como argumento.
- **ltrim:** Método privado que elimina espacios en blanco y tabuladores del inicio de una cadena pasada como argumento.
- **rtrim:** Método privado que elimina espacios en blanco y tabuladores del final de una cadena pasada como argumento.

- **strpos:** Método privado que recibe dos cadenas de caracteres. La primera es aquella en la que deseamos buscar una subcadena y la segunda, la subcadena a buscar dentro de la primera. Este método nos retorna la posición de la primera aparición de la segunda cadena dentro de la primera.
- **getIntegerValue:** Método que recibe una línea de un fichero XML y el tag a buscar en esa línea (el de inicio/final), y nos retorna el valor contenido entre los tags en formato entero. Es decir, parsea el valor de un nodo XML, y nos retorna el valor entero asociado.
- **getStrValue:** Mñetodo privado que, al igual que el anterior, busca el valor contenido en los tags de una línea de un fichero XML, y nos retorna una cadena de caracteres con dicho valor.
- **getSensorInformation:** Método privado que parsea un elemento XML de tipo 'sensor' en el fichero, y agrega su información al argumento 'CLm83List'.
- **getSensorInformation:** Método privado que parsea un elemento XML de tipo 'diode' en el fichero, y agrega su información al argumento 'CDiodeList'.
- **writeLastTemperatureXmlNode:** Método privado que genera, a partir de la información del atributo 'temperatureNode' y 'description', un nodo XML con la última temperatura leída para ese sensor/nodo en el fichero referenciado por 'fp'.
- **writeLastTemperatureXmlNode:** Método privado que genera, a partir de la información del atributo 'tempList' y 'description', un nodo XML con todas las temperaturas leídas para ese sensor/nodo en el fichero referenciado por 'fp'.

4.6.11 Clase "CI2CInterface"

CI2CInterface
#devI2c : int
+CI2CInterface() +~CI2CInterface() +open_i2c_bus() : int +read_i2c_temperature(entrada slaveAddr : unsigned int, entrada diodeSelect : unsigned char, salida temperature : unsigned char*) : int

Ilustración 40: Diagrama UML de la clase 'CI2CInterface'

4.6.11.1 Descripción

Esta clase nos ofrece un interfaz para manejar y acceder al bus de comunicaciones I2C de forma que podamos leer las temperaturas de los sensores conectados al mismo, abstrayendo a otros elementos de la aplicación de llevar a cabo estas tareas.

4.6.11.2 Atributos

- **devi2c**: Atributo protegido que contendrá el identificador del manejador del bus I2C.

4.6.11.3 Métodos

- **CI2CInterface:** Constructor de la clase sin parámetros.
- **~CI2CInterface:** Destructor de la clase.
- **open_i2c_bus:** Método público se encarga de abrir la conexión con el bus de comunicaciones I2C.
- **read_i2c_temperature:** Método público que nos permite leer una muestra de temperatura de un sensor/diodo. Para ello, recibe como argumentos la dirección del sensor LM83 con el que comunicarse, un entero que indicará si se desea leer el dato de la temperatura local del sensor LM83 o de alguno de sus diodos (‘0’ indica al sensor LM83, ‘1’ al diodo D0, ‘2’ al diodo D1 y ‘3’ al diodo D2), y un puntero a una variable de tipo unsigned char donde se almacenará el dato de temperatura leído.

4.6.12 Clase “CLogInterface”

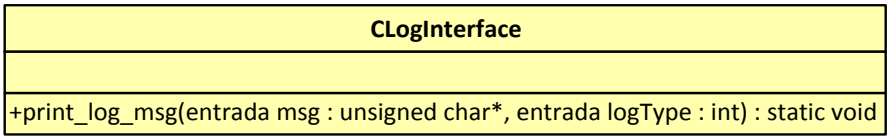


Ilustración 41: Diagrama UML de la clase 'CLogInterface'

4.6.12.1 Descripción

Esta clase nos ofrece un interfaz para manejar las capacidades de log de la aplicación. Contiene un único método que nos ofrece la funcionalidad de insertar un mensaje de log en el fichero de log del sistema operativo. Para ello, se hace uso de las capacidades del sistema ‘syslogd’ del Kernel de Linux, de forma que toda la información queda centralizada.

4.6.12.2 Métodos

- **print_log_msg:** Método estático que nos permite insertar una entrada en el log del sistema operativo. Como parámetros le daremos el mensaje a insertar, y un entero que identifica el tipo de mensaje (LOG_ERR, LOG_NOTICE, etc.).

4.6.13 Fichero principal de la aplicación: "temp_monitoring"

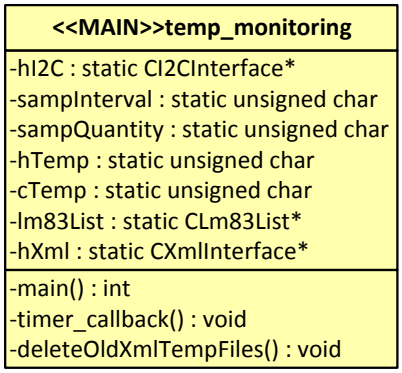


Ilustración 42: Fichero principal de la aplicación - "temp_monitoring"

4.6.13.1 Descripción

Esta clase es el cuerpo principal de la aplicación. Es decir, es la encargada de llevar a cabo todas las acciones necesarias. Se ha diseñado para ser ejecutada como un demonio del sistema operativo, de forma que se encuentre corriendo en segundo plano sin interferir con el resto de tareas. Básicamente, se encarga de inicializar los interfaces pertinentes, lee la configuración desde el fichero XML que la contiene, inicializa el bus de comunicaciones, y se dedica a leer temperaturas indefinidamente (hasta que se detenga el servicio) desde los sensores en intervalos de tiempo indicados en el fichero de configuración.

4.6.13.2 Atributos

- **hI2C**: Atributo estático que representa un manejador del bus I2C.
- **hXml**: Atributo estático que representa un manejador de fichero XML.
- **Lm83List**: Atributo estático que representa a la lista de sensores LM83 sobre la que va a trabajar la aplicación.
- **sampInterval**: Atributo estático que contiene el intervalo de muestreo de los sensores.
- **sampQuantity**: Atributo estático que contiene el número de muestras que se tomarán de cada sensor/diodo.
- **hTemp**: Atributo estático que contiene información sobre la temperatura que se considera alta.
- **cTemp**: Atributo estático que contiene información sobre la temperatura que se considera crítica.

4.6.13.3 Funciones

- **Main**: Es el punto de entrada a la aplicación, la función principal de la misma.
- **Timer_callback**: Es el callback que realiza las tareas pertinentes cada intervalo de muestreo indicado.
- **deleteOldXmlTempFiles**: Esta función se lanza al inicio de la ejecución de la aplicación, y borra los ficheros XML de temperaturas existentes (es información obsoleta) antes de comenzar a funcionar de nuevo.

4.7 Diseño de la aplicación Web para la visualización de las temperaturas muestreadas

En este apartado se expone el diseño de la aplicación Web que se encarga de mostrar al usuario la información recogida sobre las temperaturas.

4.7.1 Diseño general de la aplicación Web

La aplicación web está compuesta por un conjunto de páginas HTML, un fichero JavaScript que ofrecerá funcionalidades dinámicas, y un fichero CGI escrito en C que ampliará las capacidades de la misma para operar sobre el archivo XML de configuración. El esquema de interacción entre estos componentes (y los ficheros XML) es:

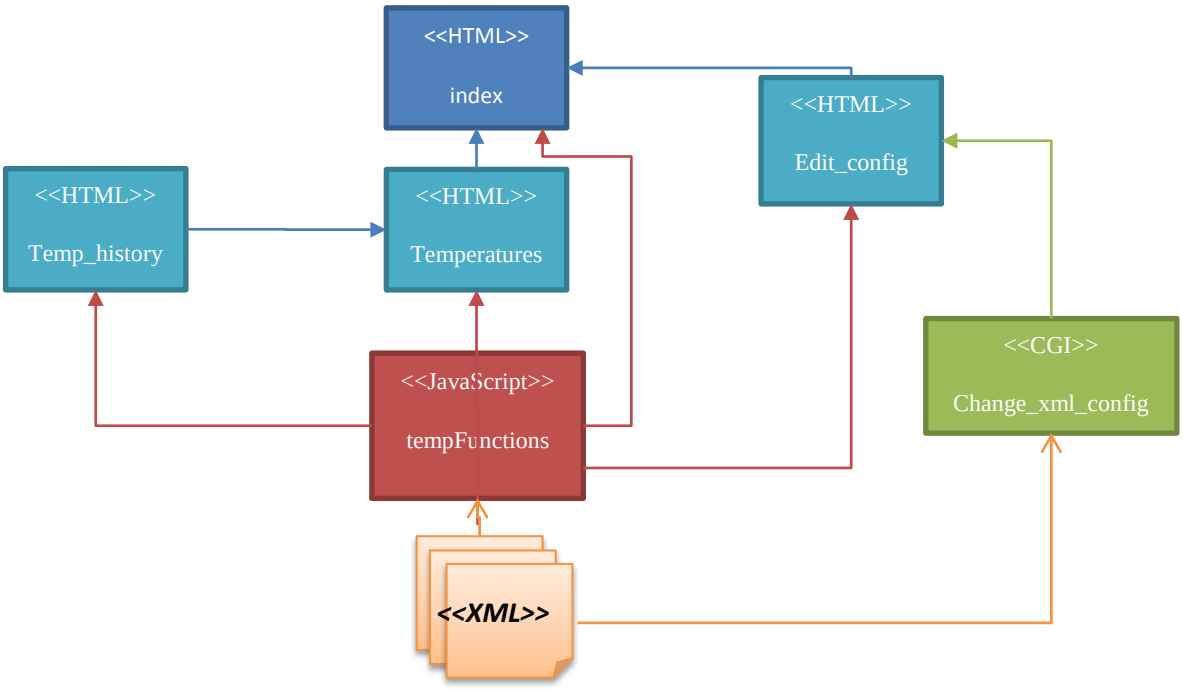


Ilustración 43: Diagrama general de la aplicación Web para la visualización de temperaturas

El punto de entrada al que accede el usuario para monitorizar las temperaturas es el módulo 'index.html'. Esta página ofrecerá dos opciones:

- **Temperature monitoring**: Esta opción, que es la mostrada por defecto, ofrecerá al usuario una visualización sobre las últimas temperaturas leídas en cada sensor. El módulo encargado de ello es 'temperatures.html'.
- **Configuration**: Este apartado ofrecerá al usuario la posibilidad de cambiar la configuración de la aplicación. Para ello, se accede a 'edit_config.html'.

Una vez que nos encontramos visualizando la última temperatura de cada sensor, es posible mostrar el histórico de las últimas temperaturas muestreadas para cada uno de ellos. Para ello, se llama al submódulo 'temp_history.html'.

Además, si accedemos a la opción que nos permite modificar la configuración, una vez llevada a cabo esta tarea se almacenará la nueva configuración descrita, para lo que se llama al módulo CGI 'change_xml_config', que actuará sobre el almacén XML que contiene la configuración de la aplicación.

Todos los HTML hacen uso del fichero JavaScript 'tempFunctions' que les ofrece funcionalidades dinámicas para operar sobre la interfaz, y por otro lado acceso a los ficheros XML de la aplicación, tanto el de configuración, como los que contienen datos sobre temperaturas.

A continuación describiremos, por un lado, el diagrama de navegación entre las distintas páginas HTML que conforman la aplicación. Después, pasaremos a mostrar los casos de uso, de forma que tengamos una visión global de como el usuario interactúa con la Web y de las opciones que tiene. Tras ello, enlazaremos esos casos de uso con las acciones que desencadenan, que pueden ser llamadas a funciones del fichero JavaScript, o llamadas al CGI. Para termina describiremos en detalle el módulo JavaScript y el CGI, que son los que aportan capacidades de ejecución dinámica a un entorno clásicamente estático como son las aplicaciones Web, dejando de lado los módulos HTML debidos a su sencillez.

4.7.2 Diagrama de navegación

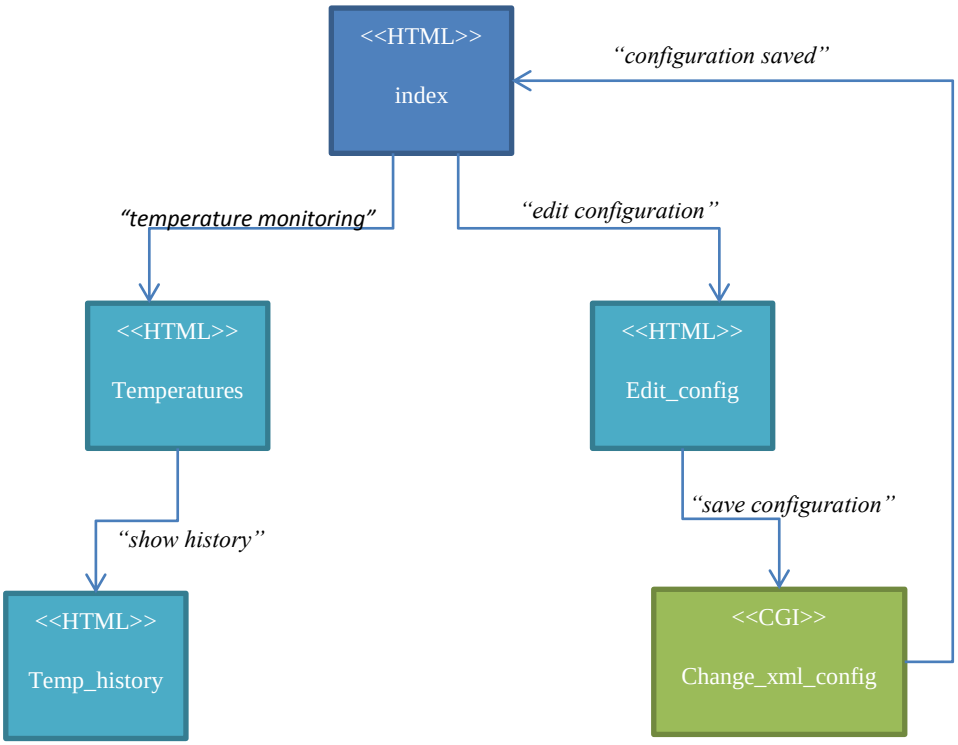


Ilustración 44: Diagrama de navegación de la aplicación Web

Las acciones que desencadenan la navegación a través de la Web son las siguientes:

- **Temperature monitoring**: Esta acción desencadena que se acceda a la página "temperatures" para monitorizar las últimas temperaturas obtenidas de los sensores.
- **Show history**: Esta acción produce que se acceda a "temp_hisotry", que mostrará el histórico de temperaturas para un sensor en concreto.
- **Edit configuration**: Esta acción produce que se acceda a la página "edit_config", que muestra el contenido del fichero XML de configuración para que el usuario pueda editarlo.
- **Save configuration:** Esta acción se produce una vez terminada la edición de la configuración, y desencadena que se llame al CGI "change_xml_config" que almacenará esta nueva configuración.
- **Configuration saved:** Tras almacenar la nueva configuración se lanza esta acción, que produce que se cargue de nuevo la página principal de la aplicación.

4.7.3 Diagrama de caso de uso

En este apartado mostraremos los diagramas de caso de uso de la aplicación. Realmente, nos encontramos ante un entorno muy sencillo, por lo que todas las opciones de interacción del usuario contra la aplicación se pueden mostrar en un único diagrama de caso de uso, que es el siguiente:

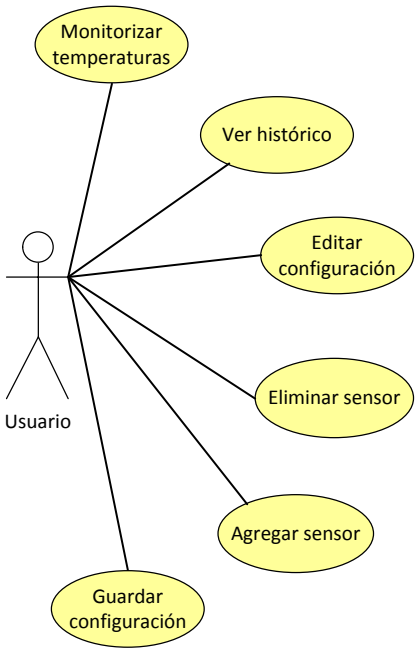


Ilustración 45: Diagrama de caso de uso de la aplicación Web

En el diagrama anterior, podemos ver que el usuario puede realizar las siguientes acciones:

- **Monitorizar temperaturas**: Esta acción dará acceso al usuario a la monitorización de la última temperatura muestreada para cada sensor. Se produce una llamada a la función "selectMenuElement('tempMenu')" del módulo 'tempFunctions' JavaScript.
- **Ver histórico**: Esta acción desplegará un gráfico con el histórico de temperaturas para un sensor/diodo en concreto. Produce una llamada a la función JavaScript 'window.open' que recibe como argumentos la URL del módulo 'temp_history' y el ID del sensor del que se desea ver el histórico.
- **Editar configuración**: Esta acción mostrará al usuario un formulario con la información de configuración de la aplicación de forma que pueda editarla. Se produce una llamada a la función "selectMenuElement('configMenu')" del módulo 'tempFunctions' JavaScript.
- **Eliminar sensor**: Esta acción provocará que se elimine la información de la configuración con respecto a un sensor. Los cambios no serán definitivos hasta que

no se guarde la nueva configuración. Produce una llamada a la función "deleteSensorFromTable" del módulo JavaScript 'tempFunctions'.

- **Agregar sensor**: Esta acción provocará que se agregue una nueva entrada para definir datos sobre un nuevo sensor en la configuración. Los cambios no serán definitivos hasta que no se guarde la nueva configuración. Produce una llamada a la función "addNewSensorToTable" del módulo JavaScript 'tempFunctions'.
- **Guardar configuración**: Esta acción provocará que los cambios realizados en la configuración se guarden de forma definitiva, generándose para ello un nuevo fichero XML. Desencadena una llamada a la función "send_new_xml_config" del módulo 'tempFunctions'.

4.7.4 Módulo JavaScript "tempFunctions"

Básicamente, este módulo aporta capacidades dinámicas a la aplicación Web. Contiene dos objetos definidos dentro del mismo y una serie de funciones que se pasan a explicar a continuación:

4.7.4.1 Objeto "Lm83Sensor"

Este objeto representa la información de un sensor LM83 leída del fichero de configuración.

4.7.4.1.1 Atributos

Los atributos de este objeto son:

- **sensorName**: Contiene el nombre del sensor.
- **sensorDescription**: Descripción del sensor.
- **sensorAddress**: Dirección del sensor.
- **D0Name**: Nombre del diodo D0.
- **D0Present**: Booleano para indicar si el diodo D0 está presente.
- **D0Description:** Descripción del diodo D0.
- **D1Name**: Nombre del diodo D1.
- **D1Present**: Booleano para indicar si el diodo D1 está presente.
- **D1Description:** Descripción del diodo D1.
- **D2Name**: Nombre del diodo D2.
- **D2Present**: Booleano para indicar si el diodo D2 está presente.
- **D2Description:** Descripción del diodo D2.

4.7.4.2 Objeto "Configuration"

Este objeto contiene la información leída desde el fichero de configuración.

4.7.4.2.1 Atributos

Los atributos del objeto son:

- **samplesQuantity**: Número de muestras a tomar de los sensores/diodos.
- **samplingInterval**: Intervalo de muestreo.
- **highTemperature**: Temperatura alta.
- **criticalTemperature**: Temperatura crítica.
- **sensorArray**: Array de sensores. Es decir, es un arreglo de objetos 'Lm83Sensor'.

4.7.4.2.2 Métodos

Los métodos de la clase son:

- **showConfig**: Este método permite formar un string con toda la información de la configuración, y mostrarla al usuario en un diálogo de alerta en la página web.

4.7.4.2.3 Funciones del módulo

A continuación podemos ver una tabla con las funciones que contiene el módulo, y una descripción del propósito de cada una:

FUNCIÓN	DESCRIPCIÓN
ajaxRequest()	Función que crea un nuevo objeto HTML de tipo 'XMLHttpRequest' y lo retorna. Este objeto se emplea para llevar a cabo peticiones asíncronas (AJAX).
selectMenuElement(elem)	Esta función se encarga de mostrar al usuario una página u otra, dependiendo de la acción que realice. El argumento que recibe indicará si se debe de obtener de forma dinámica el contenido de la página que muestra las últimas temperaturas, y el de la que permite editar la configuración de la aplicación, y lo muestra al usuario asíncronamente.
loadDivHTMLContent(url)	Función que obtiene de forma asíncrona

	el código HTML re un recurso Web y los muestra al usuario.
loadDivHTMLEditConfigContent()	Obtiene de forma asíncrona el contenido del fichero de configuración, y genera una vista HTML para mostrar esa configuración.
readConfiguration(config)	Función que obtiene el contenido del fichero "config.xml" del servidor de forma asíncrona (fichero que contiene la configuración de la aplicación)., y carga un objeto de tipo 'Configuration' con los datos leídos.
showTemperatureHistory(sensorName, config)	Función que se encarga de obtener de forma asíncrona el fichero XML que contiene en el servidor el histórico de temperaturas para un sensor en concreto, y genera el gráfico asociado para mostrar este histórico.
showLastTemperatures(config)	Función que recibe como argumento un objeto de tipo 'Configuration' y obtiene del servidor el XML que contiene las últimas temperaturas leídas, generando la interfaz HTML para mostrar esta información al usuario.
addNewSensorToTable()	Función que permite agregar a la interfaz los campos vacíos que permiten rellenar la información de un nuevo sensor para almacenarla en la configuración de la aplicación.
addSensorsToTable(config)	Función que recibe los datos de un objeto de tipo 'Configuration' e inserta en la interfaz de usuario los elementos HTML necesarios para reflejar la información relativa a los sensores.
addSensorInfoToTable(infoDiv, sensorInfo, index)	Función que recibe como argumentos la información de un sensor, el índice de ese sensor y el elemento HTML sobre el que operar, y genera el código necesario para insertar en el elemento de operación la vista necesaria para mostrar la información recibida respecto al sensor.
addNewSensorInfoToTable(infoDiv, index)	Función que genera una vista con lso campos vacíos para que puedan rellenarse y así incluir nuevos datos sobre un sensor en la configuración.
deleteSensorFromTable(index)	Función que recibe el índice HTML del sensor a borrar de la interfaz que generará la nueva configuración.
send_new_xml_config()	Función que llama al módulo CGI "change_xml_config" y le pasa la

	información de la configuración editada a partir de la cuál debe generar el nuevo fichero XML.
getUrlVars()	Función que nos permite obtener los parámetros URL recibidos en un módulo HMTL.
is_integer_valid(num, objMsg)	Función que permite validar si el dato pasado como argumento es un entero. Si no es válido, retorna un mensaje de error en el segundo parámetro.
is_hex_addr_valid(addr, objMsg)	Función que permite validar si el dato pasado como argumento es una dirección hexadecimal válida. Si no lo es, retorna un mensaje de error en el segundo parámetro.
is_str_valid(str, objMsg)	Función que permite validar si el dato pasado como argumento es una cadena válida (no vacía). Si no lo es, retorna un mensaje de error en el segundo parámetro.
window.setInterval("showLastTemperatures(conf)", (Number(conf.samplingInterval) * 1000))	Esta llamada se produce de forma automática al cargar el fichero JavaScript en el cliente. Se encarga de crear un timer que llamará a la función que muestra las últimas temperaturas cada cierto tiempo (especificado por el parámetro de periodo de muestreo definido en el fichero XML de configuración).

Tabla 36: Funciones del módulo JavaScript 'tempFunctions'

4.7.4.2.3.1 Librería 'RGraph'

Junto al módulo JavaScript, que es el encargado de las peticiones asíncronas al servidor para el manejo de datos, así como de todas las funcionalidades necesarias para operar sobre la interfaz de usuario, se ha empleado la librería 'RGraph' [DR35] para la generación de las gráficas de temperatura. Tanto las que muestran las últimas temperaturas, como el histórico de un sensor. Esta librería hace uso del estándar HTML5.

4.7.5 Módulo CGI "change_xml_config"

Este módulo recibirá una petición HTTP que contendrá información sobre la configuración de la aplicación. Su cometido será generar un nuevo fichero de configuración con los datos recibidos, de forma que se cargue la nueva configuración cuando el sistema se

reinicie y se emplee para monitorizar los sensores. Las funciones que conforman este pequeño módulo las podemos observar en la siguiente tabla:

FUNCIÓN	DESCRIPCIÓN
print_log_msg(char *msg, int logType)	Función que nos permite enviar un mensaje al entorno de log del sistema operativo.
count_html_args(char *args)	Función que se encarga de contar los argumentos recibidos en una petición HTTP.
str_to_int(char *str)	Función que lee una cadena de caracteres, y retorna el entero que representa dicha cadena.
urlDecode(char *str)	Función que decodifica los caracteres especiales recibidos en una petición HTTP convirtiéndolos a ASCII.
print_sensor_xml_information(FILE *fd, char *sName, char *sAddr, char *sDescr, char *D0Present, char *D0Descr, char *D1Present, char *D1Descr, char *D2Present, char *D2Descr)	Función que recibe como argumento la información de un sensor y el descriptor del fichero sobre el que escribir, y genera el código XML necesario para contener esos datos.
main(void)	Función principal del módulo. Recibe la petición HTTP, la decodifica y valida, y va tratando cada uno de sus argumentos para ir generando el nuevo fichero XML con la información recibida.

Tabla 37: Funciones del módulo CGI 'change_xml_config'

4.8 Prototipos de interfaces

En este apartado se muestran algunas imágenes con los prototipos de las interfaces para el módulo de visualización de temperaturas:

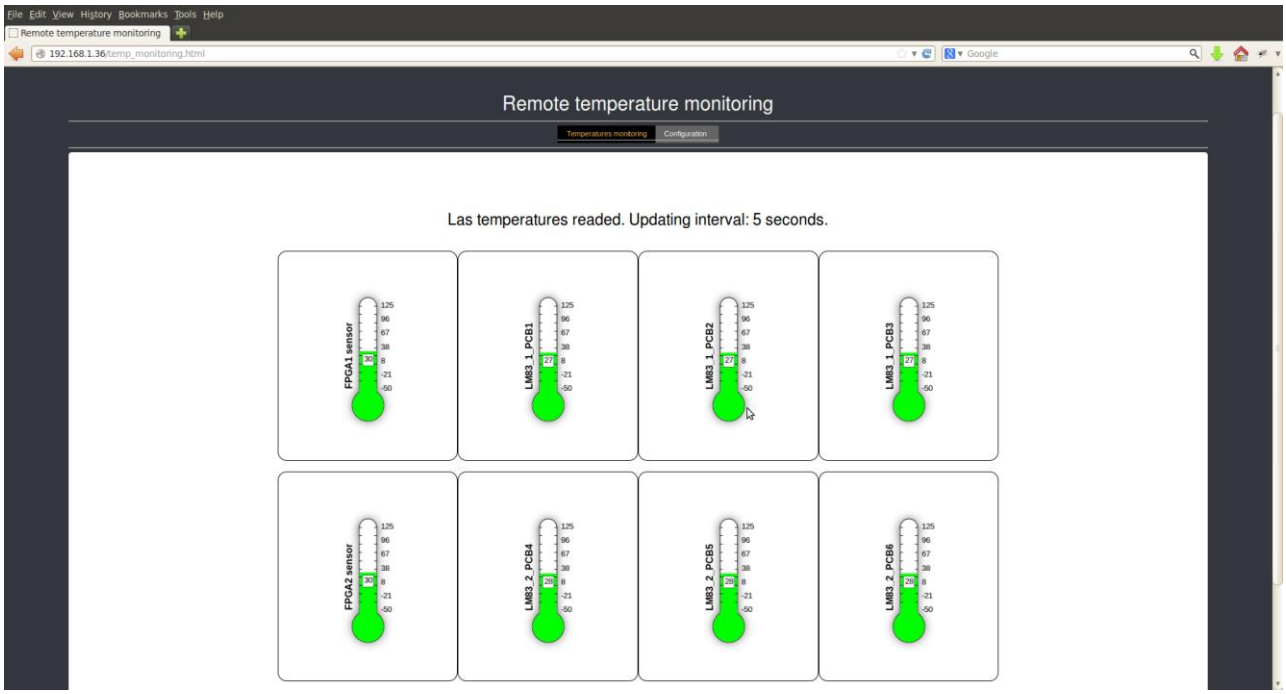


Ilustración 46: Interfaz para la monitorización de temperaturas

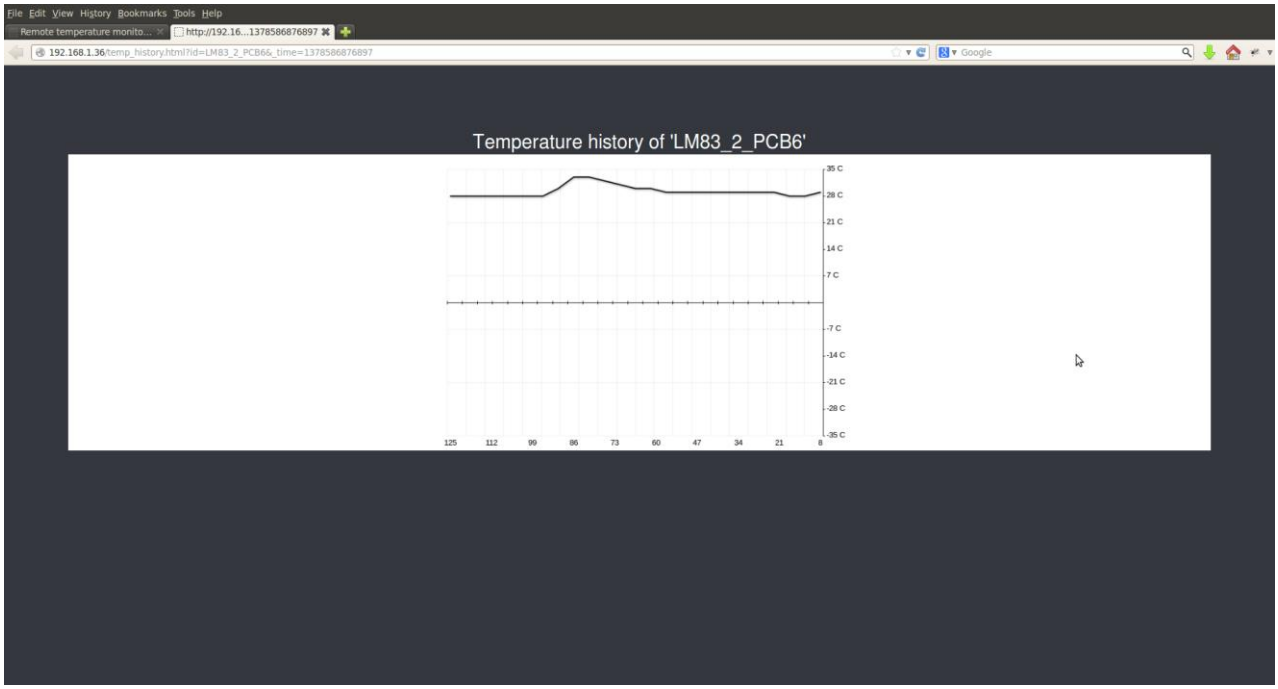


Ilustración 47:Interfaz para la visualización del histórico de temperaturas

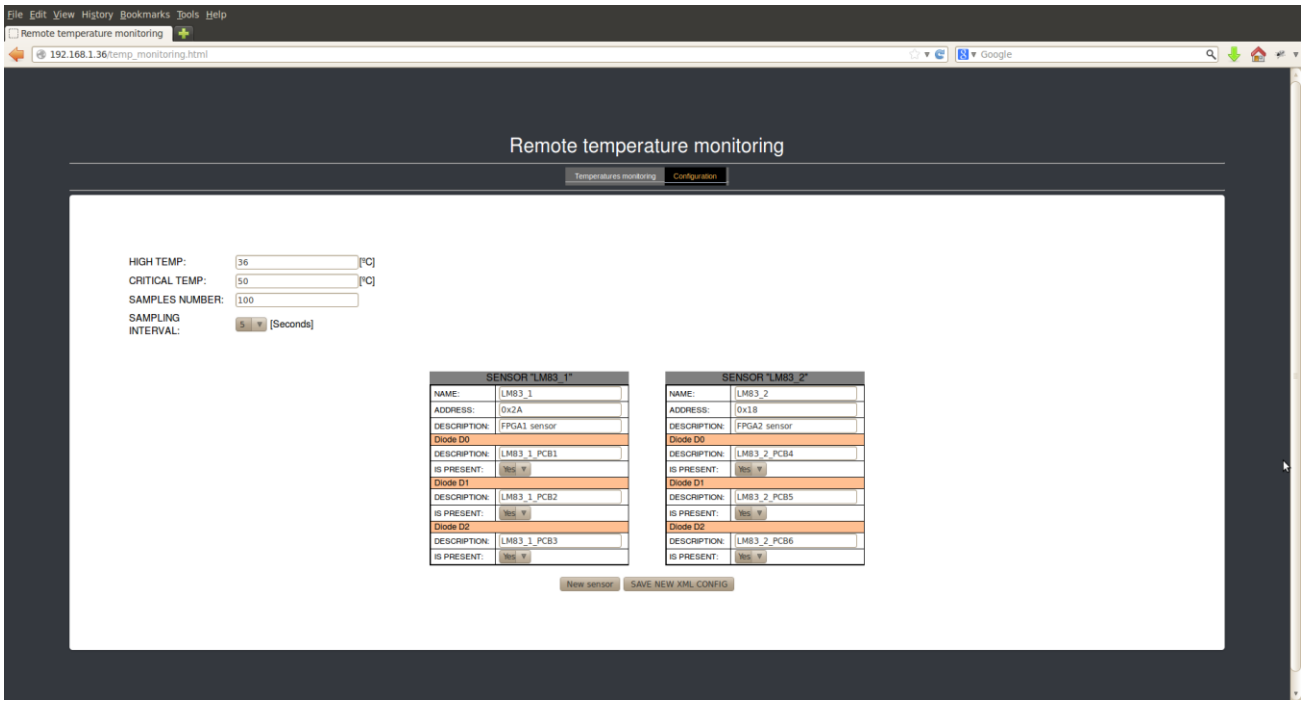


Ilustración 48: Interfaz para la modificación de la configuración de la aplicación

5 Conclusiones

En este documento se presenta el análisis, diseño y desarrollo de un prototipo de Sistema Embebido en un Chip para la monitorización remota de temperaturas. Se han expuesto los procesos involucrados en el diseño del presente prototipo, desde el montaje del circuito de pruebas para la medición de temperaturas, el diseño y configuración del Sistema Embebido en un Chip, la configuración del Soft-Core Microblaze de Xilinx para dar soporte a un sistema operativo GNU/Linux configurado y compilado para dicha arquitectura, hasta el desarrollo del Software que supone la unión de todos estos elementos para dar cabida al prototipo finalizado en todas sus etapas.

Respecto al circuito de pruebas, se ha llevado a cabo sobre una placa de prototipado, exponiendo las consideraciones más importantes sobre el mismo, así como el esquemático que puede ser empleado para, de forma sencilla, pasar a una placa de circuito impreso.

Para el diseño del Sistema Embebido en un Chip hemos empleado la herramienta EDK/XPS proporcionada por Xilinx para este propósito, creando un SoC para ser sintetizado en una placa de evaluación "*Spartan-3AN Starter Kit*" que contiene una FPGA de bajo coste de una de las familias más básicas de Xilinx. Se han expuesto así mismo los puntos clave a tener en cuenta respecto al citado diseño.

Una ven configurado el sistema embebido, se describe cómo obtener el conjunto de aplicaciones necesarias para poder desarrollar el siguiente nivel del prototipo puramente Software. Por un lado, hemos visto cómo obtener el código fuente del Kernel de GNU/Linux, así como las herramientas para la compilación cruzada ("*toolchain*"), de forma que el núcleo del sistema pueda ser configurado correctamente y compilado para ejecutarse sobre el sistema final basado en Microblaze.

Tras ello se ha expuesto una secuencia de trabajo con las consideraciones más importantes a tener en cuenta a la hora de desarrollar Software para Sistemas Embebidos. En este apartado hemos descrito cómo emplear y configurar un IDE que nos permita optimizar la forma de trabajar en estos entornos. También se exponen una serie de conceptos y las distintas posibilidades para la implementación final del Sistema operativo junto al Software que correrá sobre el mismo.

Como punto final en cuanto a la aplicación desarrollada, hemos detallado su diseño, llevando a cabo un análisis de los distintos módulos Software necesarios para su implementación, así como de las interfaces para la comunicación entre estos módulos. También se lleva a cabo una descripción de los distintos almacenes de datos que se necesitan para el correcto desarrollo del Sistema de Información diseñado.

En conclusión, este proyecto ha conseguido los objetivos que perseguía y que fueron planteados al inicio del mismo. Por un lado en la presentación del estado de arte sobre el uso de sistemas operativos de la familia GNU/Linux se ha analizado los distintos aspectos en los que se está avanzando en esta línea. A este respecto, el presente proyecto aporta un campo poco explotado, como es el uso de este tipo de SoC para la monitorización remota de

elementos en entornos donde nuestro sistema final tiene un acceso físico complicado, por lo que se puede llevar a cabo esta monitorización de forma remota.

También se ha expuesto cómo se debe configurar el Kernel de GNU/Linux para dar soporte al bus de comunicaciones I2C, así como la forma en que los drivers necesarios deben ser seleccionados para ser compilados junto al núcleo del sistema. El fabricante de sistemas reconfigurables seleccionado para desarrollar el presente prototipo (Xilinx), en sus notas técnicas, manuales y tutoriales, únicamente ha expuesto y verificado el acceso a una memoria EEPROM conectada al bus I2C, por lo que vemos como este Trabajo de Fin de Máster aporta una nueva línea de verificación del correcto funcionamiento de otros dispositivos en este bus de comunicaciones.

Otro punto que se proponía era el exponer una aplicación Web que corriese sobre un servidor Web embebido, y la definición de las interfaces entre este módulo y el encargado de monitorizar las temperaturas ofrecidas por los dispositivos conectados al bus. En este aspecto, se han implementado interfaces que generan una dependencia mínima entre ambos módulos, a la vez que se ha contrastado que el sistema en su totalidad es accesible y funciona correctamente.

En conjunto, la conclusión final es que el presente Trabajo de Fin de Máster ha verificado la posibilidad de emplear Sistemas Embebidos en entornos reconfigurables de bajo coste, aportando además información al estado del arte en este campo. Por un parte, se ha encontrado nula información sobre el despliegue de GNU/Linux en la placa de evaluación empleada (Xilinx, de nueva, únicamente ofrece información y soporte para tres modelos concretos: Spartan-6 SP605, Virtex5-ML507 y Zynq-7000 Zedboard), por lo que se ha verificado el correcto funcionamiento en un entorno completamente distinto. Además, la mayoría de la documentación oficial, tutoriales y manuales, hacen alusión a versiones del Kernel 2.6 de GNU/Linux, por lo que emplear el Kernel 3.6 ha sido otro hito conseguido que actualiza el estado del arte en este aspecto.

6 Trabajo futuro

El presente Trabajo de Fin de Máster presenta un Sistema Embebido en un Chip para la monitorización remota de temperaturas. Existen dos vías en las que podría resultar interesante el seguir trabajando en el presente proyecto, y que no han sido analizadas ni desarrolladas por incompatibilidades con la extensión y temporalidad del mismo.

6.1 Hot swapping

La primera de estas vías de ampliación consistiría en analizar y desarrollar la inclusión de "*hot swapping*" en el bus I2C. Es decir, que se puedan conectar dispositivos al bus sin tener que apagar todos los sistemas que intervienen en el mismo. Esto permitiría, por ejemplo, el agregar nuevos sensores de temperatura y que sean reconocidos y agregados a la aplicación que los sondea y a la que ofrece la información de sus temperaturas.

El problema a nivel físico de este supuesto es que el bus I2C no ha sido diseñado para esta posibilidad. La inclusión y conexión de nuevos elementos al bus puede producir transiciones en las líneas del mismo ("glitches" tanto en SDA como en SDL) que pueden resultar en inconsistencias en las comunicaciones del mismo, dando estados erróneos. Si un esclavo es conectado de repente al bus, hasta que su línea con el mismo se estabiliza, pueden producirse fluctuaciones en las señales eléctricas del mismo que hagan que las tensiones bajen (recordemos que es un bus con resistores de pull-up, por lo que su estado normal es el estar activo, y las transacciones comienzan cuando el nivel de tensión es llevado a un cero lógico), por lo que esto podría ser considerado como un inicio de transacción, y provocar errores en las comunicaciones. Incluso podría producirse una corriente excesiva o una sobretensión en las líneas del bus, por lo que algunos elementos del mismo podrían resultar dañados.

Sin embargo, existen elementos que permiten el llevar a cabo esta tarea (circuitos integrados que ofrecen facilidades de hot swapping, como por ejemplo los integrados de la familia PCA951 de NXP, TCA4311 de Texas Instruments, etc.), aislando al esclavo del bus de forma que no interfiera en las comunicaciones en curso hasta que no se finalicen, y estabilizando las líneas de comunicación antes de su conexión al bus.

Para dar soporte a esta situación, en el aspecto Software de nuestro desarrollo ya se ha incluido un avance para su implementación. La idea resulta un concepto sencillo: ya que tenemos la posibilidad de modificar la configuración de la aplicación, en la que se detallan los elementos conectados al bus, se ha llevado a cabo una modificación del módulo CGI 'change_xml_config' encargado de generar el nuevo fichero XML de comunicación. En esta aproximación, el CGI lo primero que hace es detener la aplicación de monitorización de temperaturas. Tras ello, genera el nuevo fichero XML de configuración en el que figuran los nuevos dispositivos del bus, y vuelve a arrancar el demonio para monitorizar las temperaturas. Este último, al cargar de nuevo el listado de dispositivos en el bus desde el XML generado,

sería capaz de identificar que existen nuevos sensores y, por lo tanto, sondearlos y generar los ficheros de últimas temperaturas y de histórico de las mismas correctamente.

Así pues, los pasos serían sencillos, ya que se insertaría el nuevo esclavo en el bus, se modificaría la configuración de la aplicación mediante la Web y se relanzaría el demonio de monitorización que ya sería capaz de direccionar a ese nuevo elemento del bus. Como vemos, esta línea presenta entonces un avance más de implementación Hardware analizando cómo emplear estos elementos que permiten el hot swapping y desarrollando placas de ejemplo para las pruebas que se implementación Software.

6.2 Control de la refrigeración de los distintos sistemas monitorizados

Otra línea interesante en la que avanzar en desarrollo expuesto a lo largo de esta memoria sería en el diseño y control de sistemas de refrigeración. Ya que nos encontramos monitorizando temperaturas en distintos elementos, y que tenemos definidos umbrales de temperaturas altas y críticas, podríamos ampliar las funcionalidades del prototipo para que se activasen sistemas de refrigeración bajo demanda cuando estos umbrales sean superados.

En primera instancia, existirían varias formas de abordar esta línea, como por ejemplo emplear elementos GPIO para activar/desactivar relés que alimentasen sistemas de refrigeración como podría ser un ventilador. Sin embargo, si tenemos una gran cantidad de elementos a refrigerar, el consumo de recursos físicos debido a las líneas de comunicación a manejar (una por cada sistema de refrigeración a activar/desactivar) crecería y resultaría inviable. Por ello, se ha pensado una solución que podría resultar interesante, aprovechando para ello el bus I2C que tenemos implementado.

La idea sería entonces poner estos sistemas de refrigeración como esclavos en el bus de comunicaciones I2C, de forma que puedan ser comandados para que activen/desactiven los relés de alimentación de los dispositivos que lleven a cabo la refrigeración. Una primera aproximación podría basarse en emplear FPGAs con periféricos diseñados para esta finalidad, pero este supuesto resultaría costoso debido al precio de las mismas, incluso de los modelos más sencillos. Así pues, tenemos un elemento que puede encajar perfectamente en esta idea: el microcontrolador.

Podríamos emplear estos dispositivos para conectarlos al bus I2C por un lado, y usar otra línea para manejar el relé de alimentación que active/desactive los sistemas de refrigeración. Existen modelos que ya ofrecen periféricos I2C implementados, de pequeño tamaño y bajo coste. Por ejemplo, el fabricante Microchip tiene en su catálogo los modelos PIC12F1840, PIC12LF1552 y PIC12F1822. Estos modelos son todos elementos de 8 pines (por lo tanto de pequeño tamaño y disponibles en distintos encapsulados), y con unos precios, en dólares, de 1.08-1.31$, 0.67-0.84$ y 1.00-1.20$ respectivamente. Es decir, tenemos elementos de muy bajo coste (en su precio más alto, el más sencillo que podría servirnos a la perfección

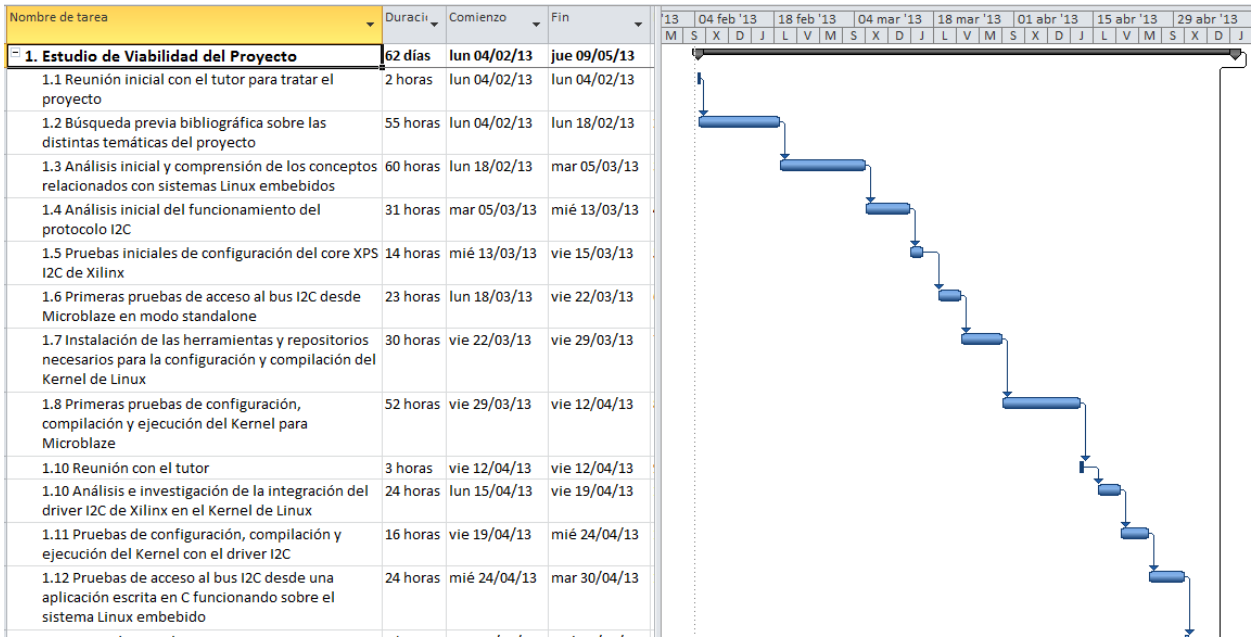
cuesta aproximadamente 0.66€) que contienen periféricos I2C ya implementados y listos para programarlos para nuestro cometido capaces de funcionar como esclavos en el bus.

Así pues, en el aspecto Hardware, habría que trabajar en la conexión de estos elementos al bus y al relé que active/desactive la refrigeración.

En el aspecto Software, estos dispositivos deberían de ser programados para recibir información por el bus, y realizar las acciones necesarias. Nuestro prototipo debería ampliar sus funcionalidades. Por un lado, reflejar en la configuración de la aplicación la dirección del esclavo en el bus que activa el sistema de refrigeración si se superan los umbrales de temperatura establecidos, y ampliar las funcionalidades del módulo que accede al bus ya no sólo para leer temperaturas, si no para comandar a estos nuevos elementos. Sin embargo, gracias al diseño modular llevado a cabo en el presente trabajo, estas modificaciones no supondrían una gran carga de trabajo.

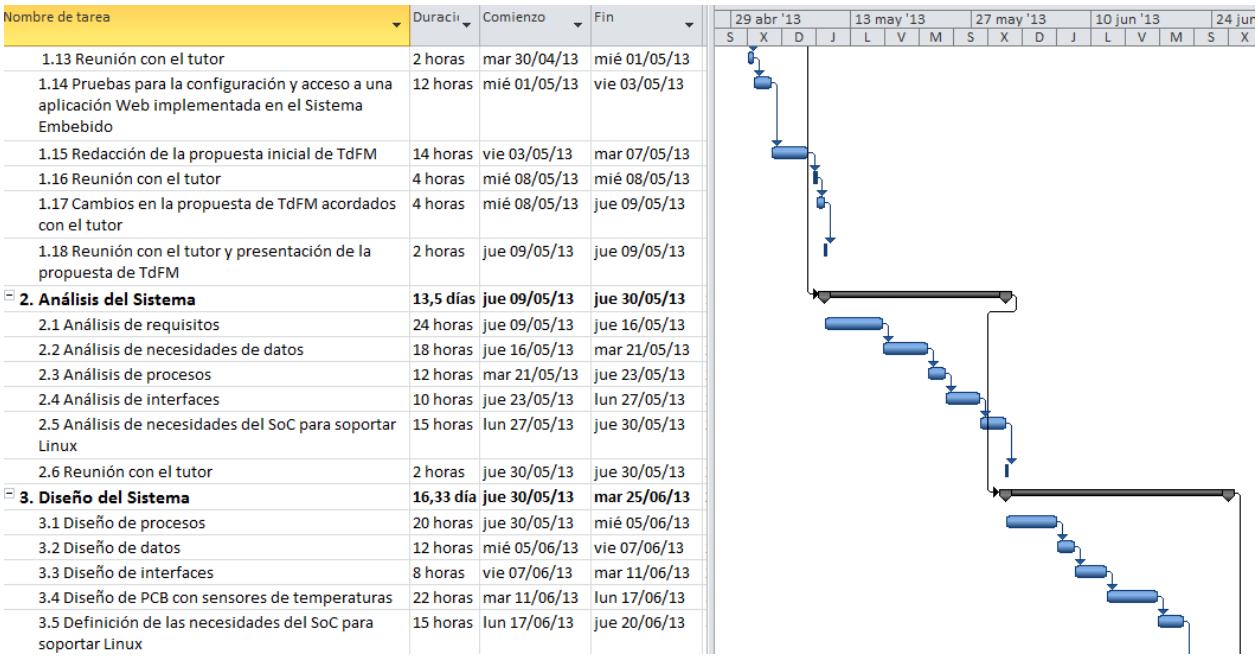
7 Planificación y dedicación al proyecto

En las siguientes imágenes podemos ver la planificación y dedicación al presente Trabajo de Fin de Máster:



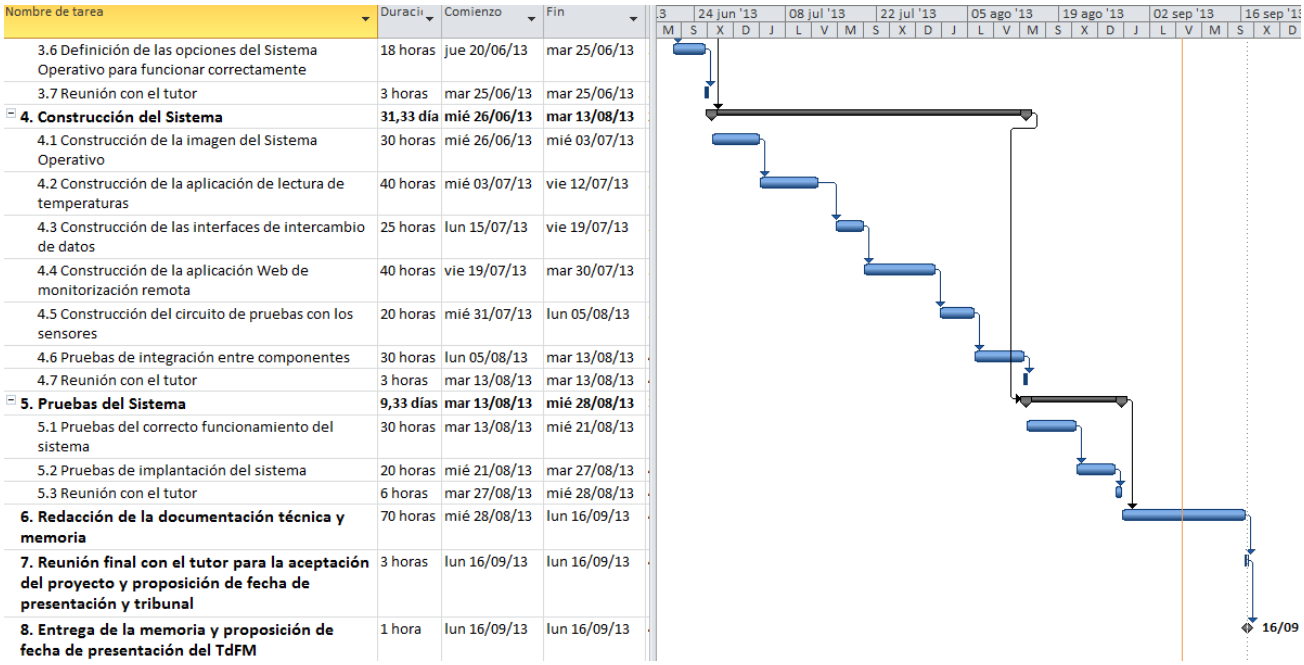
Nombre de tarea	Duraci	Comienzo	Fin
1. Estudio de Viabilidad del Proyecto	**62 días**	**lun 04/02/13**	**jue 09/05/13**
1.1 Reunión inicial con el tutor para tratar el proyecto	2 horas	lun 04/02/13	lun 04/02/13
1.2 Búsqueda previa bibliográfica sobre las distintas temáticas del proyecto	55 horas	lun 04/02/13	lun 18/02/13
1.3 Análisis inicial y comprensión de los conceptos relacionados con sistemas Linux embebidos	60 horas	lun 18/02/13	mar 05/03/13
1.4 Análisis inicial del funcionamiento del protocolo I2C	31 horas	mar 05/03/13	mié 13/03/13
1.5 Pruebas iniciales de configuración del core XPS I2C de Xilinx	14 horas	mié 13/03/13	vie 15/03/13
1.6 Primeras pruebas de acceso al bus I2C desde Microblaze en modo standalone	23 horas	lun 18/03/13	vie 22/03/13
1.7 Instalación de las herramientas y repositorios necesarios para la configuración y compilación del Kernel de Linux	30 horas	vie 22/03/13	vie 29/03/13
1.8 Primeras pruebas de configuración, compilación y ejecución del Kernel para Microblaze	52 horas	vie 29/03/13	vie 12/04/13
1.10 Reunión con el tutor	3 horas	vie 12/04/13	vie 12/04/13
1.10 Análisis e investigación de la integración del driver I2C de Xilinx en el Kernel de Linux	24 horas	lun 15/04/13	vie 19/04/13
1.11 Pruebas de configuración, compilación y ejecución del Kernel con el driver I2C	16 horas	vie 19/04/13	mié 24/04/13
1.12 Pruebas de acceso al bus I2C desde una aplicación escrita en C funcionando sobre el sistema Linux embebido	24 horas	mié 24/04/13	mar 30/04/13

Ilustración 49: Planificación del proyecto - primera parte



Nombre de tarea	Duraci	Comienzo	Fin
1.13 Reunión con el tutor	2 horas	mar 30/04/13	mié 01/05/13
1.14 Pruebas para la configuración y acceso a una aplicación Web implementada en el Sistema Embebido	12 horas	mié 01/05/13	vie 03/05/13
1.15 Redacción de la propuesta inicial de TdFM	14 horas	vie 03/05/13	mar 07/05/13
1.16 Reunión con el tutor	4 horas	mié 08/05/13	mié 08/05/13
1.17 Cambios en la propuesta de TdFM acordados con el tutor	4 horas	mié 08/05/13	jue 09/05/13
1.18 Reunión con el tutor y presentación de la propuesta de TdFM	2 horas	jue 09/05/13	jue 09/05/13
2. Análisis del Sistema	**13,5 días**	**jue 09/05/13**	**jue 30/05/13**
2.1 Análisis de requisitos	24 horas	jue 09/05/13	jue 16/05/13
2.2 Análisis de necesidades de datos	18 horas	jue 16/05/13	mar 21/05/13
2.3 Análisis de procesos	12 horas	mar 21/05/13	jue 23/05/13
2.4 Análisis de interfaces	10 horas	jue 23/05/13	lun 27/05/13
2.5 Análisis de necesidades del SoC para soportar Linux	15 horas	lun 27/05/13	jue 30/05/13
2.6 Reunión con el tutor	2 horas	jue 30/05/13	jue 30/05/13
3. Diseño del Sistema	**16,33 día**	**jue 30/05/13**	**mar 25/06/13**
3.1 Diseño de procesos	20 horas	jue 30/05/13	mié 05/06/13
3.2 Diseño de datos	12 horas	mié 05/06/13	vie 07/06/13
3.3 Diseño de interfaces	8 horas	vie 07/06/13	mar 11/06/13
3.4 Diseño de PCB con sensores de temperaturas	22 horas	mar 11/06/13	lun 17/06/13
3.5 Definición de las necesidades del SoC para soportar Linux	15 horas	lun 17/06/13	jue 20/06/13

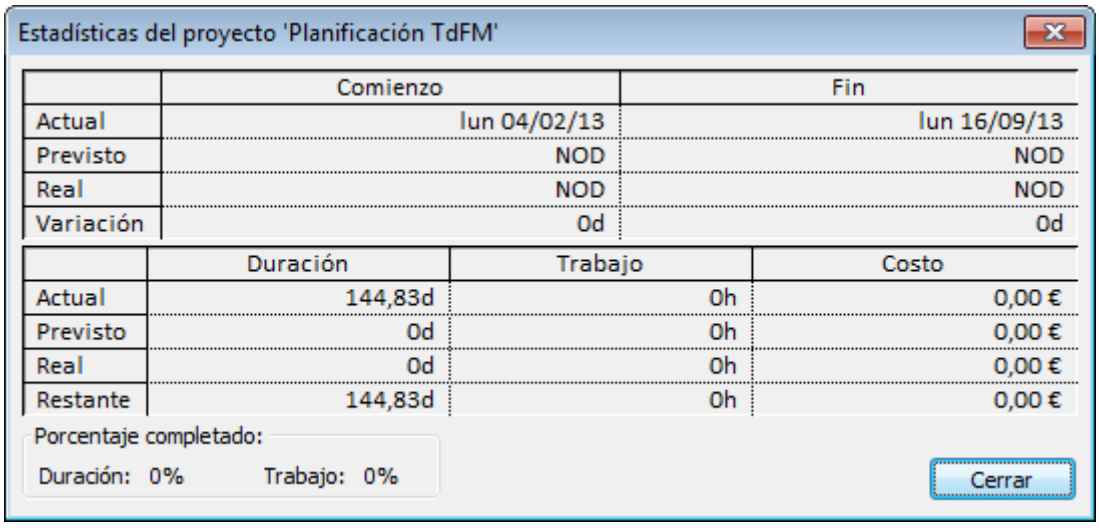
Ilustración 50: Planificación del proyecto - segunda parte



Nombre de tarea	Duración	Comienzo	Fin
3.6 Definición de las opciones del Sistema Operativo para funcionar correctamente	18 horas	jue 20/06/13	mar 25/06/13
3.7 Reunión con el tutor	3 horas	mar 25/06/13	mar 25/06/13
4. Construcción del Sistema	**31,33 día**	**mié 26/06/13**	**mar 13/08/13**
4.1 Construcción de la imagen del Sistema Operativo	30 horas	mié 26/06/13	mié 03/07/13
4.2 Construcción de la aplicación de lectura de temperaturas	40 horas	mié 03/07/13	vie 12/07/13
4.3 Construcción de las interfaces de intercambio de datos	25 horas	lun 15/07/13	vie 19/07/13
4.4 Construcción de la aplicación Web de monitorización remota	40 horas	vie 19/07/13	mar 30/07/13
4.5 Construcción del circuito de pruebas con los sensores	20 horas	mié 31/07/13	lun 05/08/13
4.6 Pruebas de integración entre componentes	30 horas	lun 05/08/13	mar 13/08/13
4.7 Reunión con el tutor	3 horas	mar 13/08/13	mar 13/08/13
5. Pruebas del Sistema	**9,33 días**	**mar 13/08/13**	**mié 28/08/13**
5.1 Pruebas del correcto funcionamiento del sistema	30 horas	mar 13/08/13	mié 21/08/13
5.2 Pruebas de implantación del sistema	20 horas	mié 21/08/13	mar 27/08/13
5.3 Reunión con el tutor	6 horas	mar 27/08/13	mié 28/08/13
6. Redacción de la documentación técnica y memoria	70 horas	mié 28/08/13	lun 16/09/13
7. Reunión final con el tutor para la aceptación del proyecto y proposición de fecha de presentación y tribunal	3 horas	lun 16/09/13	lun 16/09/13
8. Entrega de la memoria y proposición de fecha de presentación del TdFM	1 hora	lun 16/09/13	lun 16/09/13

Ilustración 51: Planificación del proyecto - tercera parte

En la siguiente imagen, además, podemos ver un resumen del proyecto:



Estadísticas del proyecto 'Planificación TdFM'

	Comienzo	Fin
Actual	lun 04/02/13	lun 16/09/13
Previsto	NOD	NOD
Real	NOD	NOD
Variación	0d	0d

	Duración	Trabajo	Costo
Actual	144,83d	0h	0,00 €
Previsto	0d	0h	0,00 €
Real	0d	0h	0,00 €
Restante	144,83d	0h	0,00 €

Porcentaje completado:
Duración: 0% Trabajo: 0%
Cerrar

Ilustración 52: Estadísticas de la planificación del proyecto

Como vemos, se tiene una dedicación de 145 días aproximadamente. Teniendo en cuenta que el calendario laboral generado para la planificación es de 5 horas al día (17:30 - 21:30 p.m.), se han empleado **725 horas** en la realización del proyecto, algo más de lo planificado inicialmente al presentar la propuesta de proyecto.

8 Referencias bibliográficas

[DR1] Xilinx Inc., 2011, “Extended SPARTAN-3A Family Overview” [Online], Disponible en http://www.xilinx.com/support/documentation/data_sheets/ds706.pdf [Último acceso: 02-05-2013].

[DR2] Xilinx Inc., 2008, “Spartan-3A/AN Starter Kit Board User’s Guide” [Online], Disponible en <http://www.xilinx.com/support/documentation/boards_and_kits/ug334.pdf> [Último acceso: 03-05-2013].

[DR3] Zhou Qingguo; Yao Qi; Li Chanjuan; Hu Bin, "Port embedded Linux to XUP Virtex-II Pro development board," IT in Medicine & Education, 2009. ITIME '09. IEEE International Symposium on , vol.1, no., pp.165,169, 14-16 Aug. 2009.

[DR4] Litayem, N.; Ghrissi, M.; Ben Salem, A.K.; Ben Saoud, S., "Designing and building embedded environment for robotic control application," Industrial Electronics, 2009. IECON '09. 35th Annual Conference of IEEE, vol., no., pp.2907,2912, 3-5 Nov. 2009.

[DR5] Patel, R.; Rajawat, A.; Yadav, R. N., "Remote Access of Peripherals Using Web Server on FPGA Platform," Recent Trends in Information, Telecommunication and Computing (ITC), 2010 International Conference on , vol., no., pp.274,276, 12-13 March 2010.

[DR6] Xilinx Inc., “Xilinx Open Source Linux Wiki – MicroBlaze Linux (General)” [Online], Disponible en <http://xilinx.wikidot.com/microblaze-linux> [Último acceso: 01-05-2013].

[DR7] Billauer, E., 2011, “Linux on Microblaze HOWTO” [Online], Disponible en http://billauer.co.il/blog/2011/08/linux-microblaze-howto-tutorial-primer-1/ [Último acceso: 05-05-2013].

[DR8] Kebschull, U., Kugel, A., 2013, “Tutorial: Linux on Microblaze” [Online], Dpto. para la aplicación de computación científica, Universidad de Heidelberg, Disponible en http://li5.ziti.uni-heidelberg.de/lectures/2010hws/embedded/linuxTutorial.pdf [Último acceso: 05-05-2013].

[DR9] Coxe, R., 2011, “Linux Port to a Microblaze on a Xilinx ML605 FPGA board” [Online], Close-Haul Communications, USA, Disponible en <http://www.closehaul.com/2011/09/linux-port-to-a-microblaze-on-a-xilinx-ml605-fpga-board/>[Último acceso: 05-05-2013].

[DR10] Xilinx Inc., 2012, “MicroBlaze Processor Reference Guide” [Online], Disponible en <http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_1/mb_ref_guide.pdf> [Último acceso: 05-05-2013].

[DR11] Xilinx Inc., 2012, “OS and Libraries Document Collection” [Online], Disponible en <http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_3/oslib_rm.pdf> [Último acceso: 05-05-2013].

[DR12] Barr, M., Massa, A., 2006, "Programming Embedded Systems, 2nd Edition, with C and GNU Development Tools", 2nd Edition, O'Rilley.

[DR13] NXP Semiconductors, 2012, "I²C-bus specification and user manual" [Online], Disponible en http://www.nxp.com/documents/user_manual/UM10204.pdf [Último acceso: 05-05-2013].

[DR14] Corbert, J., Rubini, A., Kroah-Hartman, G., 2005, "Linux Device Drivers", 3rd Ed., O'Reilly.

[DR15] Yaghmour, K., Masters, J., Ben-Yossef, G., Gerum, P., 2008, "Building Embedded Linux Systems", 2nd Ed., O'Reilly.

[DR16] "20 Years of FPGA Evolution from Glue Logic to Major System Component", Perter Alfke, Xilinx. PDF disponible en "http://www.hotchips.org/wp-content/uploads/hc_archives/hc19/1_Sun/HC19.tutorial1.02.pdf". Última visita [10-08-2013].

[DR17] Xilinx Inc., "Microblaze Linux (General)" [Online], Disponible en: <http://xilinx.wikidot.com/microblaze-linux> [Último acceso: 02-09-2013]

[DR18] Kernel Newbies, "Linux 2 6 31" [Online], Disponible en: <http://kernelnewbies.org/Linux_2_6_31> [Último acceso: 12-09-2013]

[DR19] Petalogix, "Overview of PetaLinux" [Online], Disponible en <http://www.petalogix.com/resources/documentation/petalinux/userguide/Overview> [Último acceso: 05-05-2013].

[DR20] Xilinx Inc., "Linux I2C Driver" [Online], Disponible en <http://www.wiki.xilinx.com/Linux+I2C+Driver> [Último acceso: 05-05-2013].

[DR21] Sanchez, A.R.; Alvarado-Nava, O.; Martinez, F.J.Z., "Network monitoring system based on an FPGA with Linux," Technologies Applied to Electronics Teaching (TAEE), 2012, vol., no., pp.232,236, 13-15 June 2012.

[DR22] Moldovan, I.; Varga, P., "A flexible switch-router with reconfigurable forwarding and Linux-based Control Element," Electronics and Telecommunications (ISETC), 2012 10th International Symposium on, vol., no., pp.217,220, 15-16 Nov. 2012.

[DR23] Guoqin Wang; Meihua Xu; Xuan Huan, "Design and implementation of an embedded router with packet filtering," Electrical & Electronics Engineering (EEESYM), 2012 IEEE Symposium on , vol., no., pp.285,288, 24-27 June 2012.

[DR24] Lin Zhang; Slaets, P.; Bruyninckx, H., "An open embedded hardware and software architecture applied to industrial robot control," Mechatronics and Automation (ICMA), 2012 International Conference on, vol., no., pp.1822,1828, 5-8 Aug. 2012.

[DR25] Hagiwara, H.; Asami, K.; Komori, M., "Real-time image processing system by using FPGA for service robots," Consumer Electronics (GCCE), 2012 IEEE 1st Global Conference on, vol., no., pp.720,723, 2-5 Oct. 2012.

[DR26] Hayek, Ali; Domes, Sebastian; Borcsok, Josef, "Internet-controlled dynamic reconfiguration for FPGA-based embedded systems," Communications and Information Technology (ICCIT), 2013 Third International Conference on , vol., no., pp.190,194, 19-21 June 2013.

[DR27] Mendon, A.A.; Bin Huang; Sass, R., "A high performance, open source SATA2 core," Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on , vol., no., pp.421,428, 29-31 Aug. 2012.

[DR28] Diolan, "I2C Bus Specification" [Online], Disponible en: <http://www.diolan.com/dln_doc/i2c-bus-specification.html> [Último acceso: 05-09-2013].

[DR29] Texas Instruments, "LM83. Triple-Diode Input and Local Digital Temperature Sensor with Two-Wire Interface" [Online], Disponible en: <http://www.ti.com/lit/ds/snis111a/snis111a.pdf> [Último acces0: 10-09-2013].

[DR30] Colin, A., "SettingUpNFSHowTo" [Online], 2013, Disponible en: <https://help.ubuntu.com/community/SettingUpNFSHowTo> [Último acceso: 12-07-2013].

[DR31] http://www.soportejm.com.sv/kb/index.php/article/lucid-lynx-nfs

[DR32] Watson, D., "Linux Daemon Writing HOWTO" [Online], Disponible en: <http://www.netzmafia.de/skripten/unix/linux-daemon-howto.html> [Último acceso: 07-07-2013].

[DR33] Márquez, F. M., 2004, "UNIX. Programación Avanzada. 3ª Edición ampliada y actualizada", 3rd Ed., Ra-Ma.

[DR34] Stackoverflow, "Ubuntu Linux C++ error: undefined reference to 'clock_gettime' and 'clock_settime'" [Online] , Disponible en : <http://stackoverflow.com/questions/2418157/ubuntu-linux-c-error-undefined-reference-to-clock-gettime-and-clock-settim> [Último acceso: 12-06-2013].

[DR35] Heyes, R., "RGraph – HTML5 and JavaScript charts" [Online], Disopnible en: <http://www.rgraph.net/> [Último acceso: 21-08-2013

Anexo I: Agregando un reloj en tiempo real (RTC) al prototipo

Por motivos temporales y de finalización de la redacción de la presente memoria este apartado no se ha incluido en el diseño original. Sin embargo, en la semana precedente a la entrega de la memoria, se ha realizado una ampliación del diseño que resulta interesante, por lo que se ha decido exponerla en este anexo.

La citada ampliación consiste en agregar un reloj en tiempo real al prototipo diseñado para este Trabajo de Fin de Máster. Este reloj es el dispositivo M41T00CAP del fabricante ST Microelectronics. Este integrado consiste en un reloj en tiempo real con una batería interna, lo que le permite mantener la fecha y seguir la cuenta independientemente de que se deje de alimentar al mismo. Además, este dispositivo es accesible a través del bus I2C, por lo que quedará integrado en nuestro diseño de manera que el Kernel lo reconozca durante el arranque, y acceda a él para leer la fecha y la hora, y establecer esta información en Sistema Operativo de nuestro prototipo.

Para ello, una vez conectado el dispositivo al bus I2C, debemos llevar a cabo dos pasos. El primero es editar el fichero DTS que hemos generado para nuestro SoC. En el apartado que describe el periférico I2C, agregaremos una entrada que le indicará al Sistema Operativo durante su arranque que existe un reloj en tiempo real conectado al bus I2C. Este paso lo llevamos a cabo modificando la entrada como podemos ver a continuación:

```
xps_iic_0: i2c@81600000 {
	compatible = "xlnx,xps-iic-2.03.a", "xlnx,xps-iic-2.00.a";
	interrupt-parent = <&xps_intc_0>;
	interrupts = < 0 2 >;
	reg = < 0x81600000 0x10000 >;
	xlnx,clk-freq = <0x3b9aca0>;
	xlnx,family = "spartan3a";
	xlnx,gpo-width = <0x1>;
	xlnx,iic-freq = <0x186a0>;
	xlnx,scl-inertial-delay = <0x5>;
	xlnx,sda-inertial-delay = <0x5>;
	xlnx,ten-bit-adr = <0x0>;
	#address-cells = <1>;
	#size-cells = <0>;

	rtc@68{
		compatible = “st,m41t00”;
		reg = <0x68>;
	}
} ;
```

Tabla 38: Modificando el DTS para agregar un RTC conectado al bus I2C

Con esta modificación, le indicamos la disponibilidad del RTC en el bus, además de la dirección a través de la que se comunicará con él (0x68). Además, deberemos configurar el Kernel para que se agreguen durante la compilación las facilidades RTC, y activar también el soporte para este dispositivo. Para ello, configuraremos las siguientes opciones del Kernel:

1. **Device Drivers → Real Time Clock**: Habilitaremos esta opción, lo cual nos permitirá configurar otros elementos.
2. **Device Drivers → Real Time Clock → /sys/class/rtc/rtcN:** Desactivaremos esta opción.
3. **Device Drivers → Real Time Clock → /proc/driver/rtc:** Desactivamos este elemento.
4. **Device Drivers → Real Time Clock → /dev/rtcN:** Nos aseguramos de que este elemento esté marcado como parte del Kernel.
5. **Device Drivers → Real Time Clock → Set system time from RTC on startup and resume:** Esta opción también debe estar marcada como parte del Kernel.
6. **Device Drivers → Real Time Clock → Dallas/Maxim DS1307/37/38/39/40, ST M41T00, EPSON RX-8025:** Esta opción la marcamos para su compilación como parte del Kernel.

Tras activar estas opciones, debemos recompilar de nuevo nuestro núcleo para que sea capaz de reconocer las modificaciones. Una vez hecho esto y arrancado el sistema, podemos ver en el log de inicio como la fecha es leída correctamente:

```
Early console on uartlite at 0x84000000
bootconsole [earlyser0] enabled
Ramdisk addr 0x0000003f, Compiled-in FDT at 0xc0293bb0
Linux version 3.6.0-dirty (lemito@lemito-desktop-ubuntu) (gcc version 4.6.2
20111018 (prerelease) (crosstool-NG 1.14.1) ) #1 Sat Aug 31 16:11:03 CEST 2013
setup_cpuinfo: initialising
setup_cpuinfo: Using full CPU PVR support
cache: wt_msr_noirq
setup_memory: max_mapnr: 0x4000
setup_memory: min_low_pfn: 0x44000
setup_memory: max_low_pfn: 0x48000
setup_memory: max_pfn: 0x48000
Zone ranges:
  DMA      [mem 0x44000000-0x47ffffff]
  Normal   empty
Movable zone start for each node
Early memory node ranges
  node   0: [mem 0x44000000-0x47ffffff]
On node 0 totalpages: 16384
free_area_init_node: node 0, pgdat c03590ac, node_mem_map c03a6000
  DMA zone: 128 pages used for memmap
  DMA zone: 0 pages reserved
  DMA zone: 16256 pages, LIFO batch:3
early_printk_console remapping from 0x84000000 to 0xffffd000
pcpu-alloc: s0 r0 d32768 u32768 alloc=1*32768
pcpu-alloc: [0] 0
Built 1 zonelists in Zone order, mobility grouping on.  Total pages: 16256
Kernel command line: console=ttyUL0,115200 ip=on rootfstype=nfs root=/dev/nfs rw
nfsroot=192.168.1.35:/shared/nfsroot2,tcp
PID hash table entries: 256 (order: -2, 1024 bytes)
Dentry cache hash table entries: 8192 (order: 3, 32768 bytes)
Inode-cache hash table entries: 4096 (order: 2, 16384 bytes)
Memory: 61128k/65536k available (3316k kernel code, 4408k reserved, 114k data,
169k bss, 120k init)
```

```
Kernel virtual memory layout:
  * 0xfffe000..0xffff000  : fixmap
  * 0xffffd000..0xfffe000  : early ioremap
  * 0xf0000000..0xffffd000  : vmalloc & ioremap
NR_IRQS:33
interrupt-controller #0 at 0xf0000000, num_irq=4, edge=0xe
No chosen timer found, using default
timer #0 at 0xf0002000, irq=1
microblaze_timer_set_mode: shutdown
microblaze_timer_set_mode: periodic
Calibrating delay loop... 30.10 BogoMIPS (lpj=150528)
pid_max: default: 4096 minimum: 301
Mount-cache hash table entries: 512
NET: Registered protocol family 16
i2c-core: driver [dummy] registered
bio: create slab <bio-0> at 0
Switching to clocksource microblaze_clocksource
NET: Registered protocol family 2
TCP established hash table entries: 2048 (order: 2, 16384 bytes)
TCP bind hash table entries: 2048 (order: 3, 40960 bytes)
TCP: Hash tables configured (established 2048 bind 2048)
TCP: reno registered
UDP hash table entries: 128 (order: 0, 6144 bytes)
UDP-Lite hash table entries: 128 (order: 0, 6144 bytes)
NET: Registered protocol family 1
RPC: Registered named UNIX socket transport module.
RPC: Registered udp transport module.
RPC: Registered tcp transport module.
RPC: Registered tcp NFSv4.1 backchannel transport module.
NFS: Registering the id_resolver key type
Key type id_resolver registered
Key type id_legacy registered
msgmni has been set to 119
io scheduler noop registered
io scheduler deadline registered
io scheduler cfq registered (default)
84000000.serial: ttyUL0 at MMIO 0x84000000 (irq = 3) is a uartlite
console [ttyUL0] enabled, bootconsole disabled
console [ttyUL0] enabled, bootconsole disabled
uartlite 84400000.serial: failed to get alias id, errno -19
uartlite 84400000.serial: failed to get port-number
brd: module loaded
xilinx_emaclite 81000000.ethernet: Device Tree Probing
No mdio handler
xilinx_emaclite 81000000.ethernet: error registering MDIO bus
xilinx_emaclite 81000000.ethernet: MAC address is now 00:0a:35:00:00:00
xilinx_emaclite  81000000.ethernet:  Xilinx  EmacLite  at  0x81000000  mapped  to
0xF0020000, irq=2
i2c-core: driver [rtc-ds1307] registered
i2c /dev entries driver
Device Tree Probing 'i2c'
i2c-dev: adapter [xilinx-iic] registered as minor 0
i2c i2c-0: adapter [xilinx-iic] registered
xilinx-iic #0 at 0x81600000 mapped to 0xF0040000, irq=4
rtc-ds1307 0-0068: probe
i2c i2c-0: master_xfer[0] W, addr=0x68, len=1
i2c i2c-0: master_xfer[1] R, addr=0x68, len=8
i2c i2c-0: master_xfer[0] W, addr=0x68, len=1
i2c i2c-0: master_xfer[1] R, addr=0x68, len=7
```

```
rtc-ds1307 0-0068: rtc core: registered m41t00 as rtc0
i2c i2c-0: client [m41t00] registered with bus id 0-0068
TCP: cubic registered
NET: Registered protocol family 17
Key type dns_resolver registered
i2c i2c-0: master_xfer[0] W, addr=0x68, len=1
i2c i2c-0: master_xfer[1] R, addr=0x68, len=7
rtc-ds1307 0-0068: setting system clock to 2013-09-07 23:27:26 UTC (1378596446)
Sending DHCP requests ., OK
IP-Config: Got DHCP answer from 192.168.1.1, my address is 192.168.1.36
IP-Config: Complete:
     device=eth0, addr=192.168.1.36, mask=255.255.255.0, gw=192.168.1.1
     host=dhcppc3, domain=, nis-domain=(none)
     bootserver=192.168.1.1, rootserver=192.168.1.35, rootpath=
VFS: Mounted root (nfs filesystem) on device 0:9.
Freeing unused kernel memory: 120k freed
Starting rcS...
++ Creating device points
mkdir: cannot create directory '/dev/pts': File exists
++ Mounting filesystem
++ Loading system loggers
++ Starting telnet daemon
rcS Complete
/bin/sh: can't access tty; job control turned off
/ #
```

Tabla 39: Log de inicio con el establecimiento de la fecha y hora a través de un RTC

Y una imagen de la placa de prototipado modificada para incluir este nuevo elemento en el circuito podemos ver la a continuación:

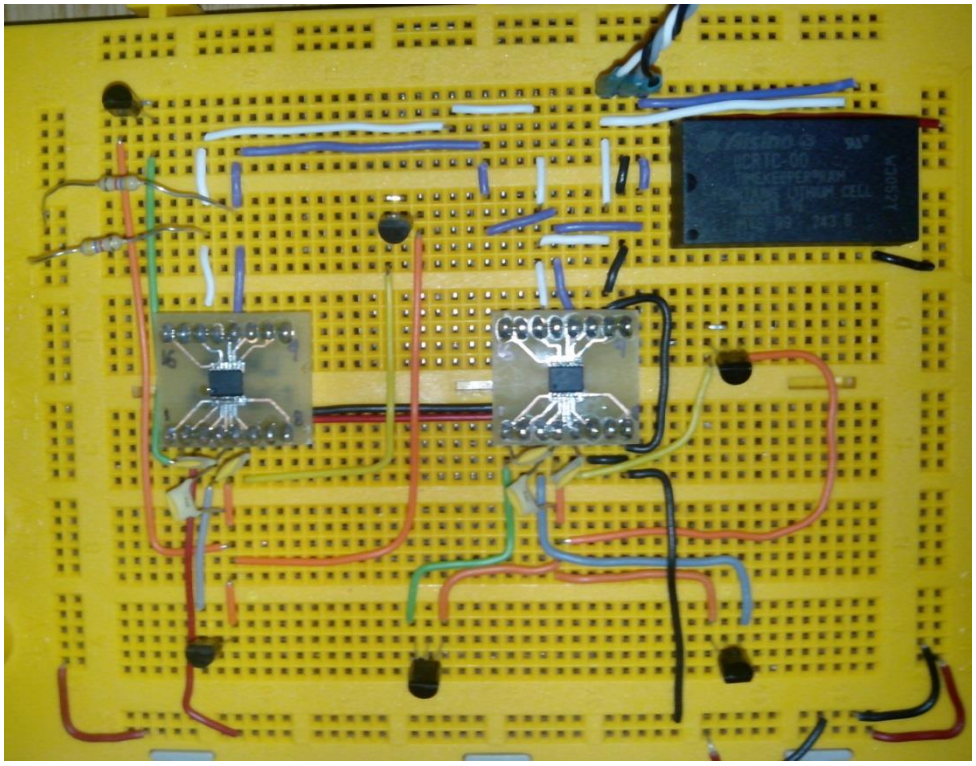


Ilustración 53: Placa de prototipado con RTC incluido

Problema debido a la inclusión del reloj RTC en el sistema

Agregar el reloj RTC al sistema hizo que apareciera un problema. Anteriormente, al crear ficheros en el sistema (por ejemplo, la aplicación de monitorización con los XML de las temperaturas), la fecha de creación de los mismos era antigua, pues se empleaba la fecha por defecto que establece el Kernel si no tiene desde donde actualizarla. Sin embargo, al añadir las facilidades del RTC, estas fechas de creación eran actuales. Esto conlleva un problema con las cachés HTTP. Los navegadores almacenan en una caché las últimas peticiones que realizas, de forma que si se vuelven a pedir esos recursos y la fecha en que se obtuvieron desde el servidor remoto es reciente, se sirve esa copia en vez de traer la información de nuevo desde el servidor.

Para nuestro diseño esto es un problema, porque las peticiones de los ficheros XML que contienen los datos de temperatura quedan cacheadas, y se nos devuelve esa copia obsoleta en vez de los nuevos datos de temperatura muestreados. Sin el RTC esto no pasaba por que la información siempre tenía una fecha muy antigua, y siempre se volvía a solicitar al considerarla obsoleta. La forma en que se ha solucionado este inconveniente es sencilla: ya que las peticiones se manejan mediante el módulo JavaScript, se ha agregado un parámetro final a cada petición HTTP que contiene una marca temporal, basada en el tiempo obtenido del sistema. Esto hace que la petición HTTP generada sea cada vez distinta al contener un parámetro que cuyo valor no coincide nunca, por lo que el navegador siempre volverá a solicitar la información al servidor en vez de devolvernos la que tiene en caché.